COURSE MATERIAL

II Year B. Tech II- Semester

MECHANICAL ENGINEERING

AY: 2022-23

INTRODUCTION TO DBMS

R20A0551

Prepared by:
Dr. R. Hussain Vali
Assistant Professor

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF MECHANICAL ENGINEERING

(Autonomous Institution-UGC, Govt. of India)
Secunderabad-500100, Telangana State, India.
www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech. ME- II Sem **L/T/P/C**
3/-/-/3

OPEN ELECTIVE - I
(R20A0551) INTRODUCTION TO DBMS

COURSE OBJECTIVES:

1) To understand the basic concepts and the applications of database systems
2) To Master the basics of SQL and construct queries using SQL
3) To understand the relational database design principles
4) To become familiar with the basic issues of transaction processing and concurrency control
5) To 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 covers-Definitions 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 Timestamps, 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 Edition 2007.
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.

COURSE OUTCOMES:

1) Understand the basic concepts and the applications of database systems
2) Master the basics of SQL and construct queries using SQL
3) Understand the relational database design principles
4) Familiarize with the basic issues of transaction processing and concurrency control
5) Familiarize with database storage structures and access techniques

UNIT I

INTRODUCTION

DATABASE

Data:

It is a collection of information.

The facts that can be recorded and which have implicit meaning known as 'data'.

Example:

Customer -----

1.cname.

2.cno.

3.ccity.

Database:

- It is a collection of interrelated data.
- These can be stored in the form of tables.
- A database can be of any size and varying complexity.
- A database may be generated and manipulated manually or it may be computerized.

Example:

Customer database consists the fields as cname, cno, and ccity

Cname	Cno	Ccity

Database System:

It is computerized system, whose overall purpose is to maintain the information and to make that the information is available on demand.

Advantages:

1.Redundency can be reduced

2.Inconsistency can be avoided.

3.Data can be shared

4.Standards can be enforced.

5.Security restrictions can be applied.

6.Integrity can be maintained.

7.Data gathering can be possible.

8.Requirements can be balanced.

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

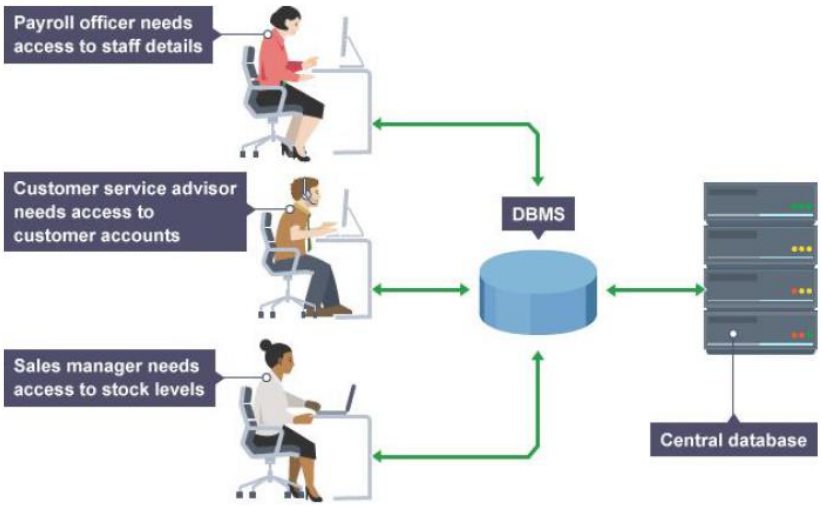


Figure: Employees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to see different 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.

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.

Database Management System (DBMS):

It is a collection of programs that enables user to create and maintain a database. In other words it is general-purpose software that provides the users with the processes of defining, constructing and manipulating the database for various applications.

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 generation of pay checks.

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

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

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

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

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

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

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an 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

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 in several 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 of the same data may no longer agree.

For example, a changed student address may be reflected in the Music department 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 no application program on hand to meet it. There is, however, an application program to generate the list of *all* students.

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

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 department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

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.

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 programs run 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 programs that have not been coordinated previously.

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 the file-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 be carried 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 continues 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

- It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.
- Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.
- 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.
- 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.

People who deal with databases

Many persons are involved in the design, use and maintenance of any database. These persons can be classified into 2 types as below.

Actors on the scene:

The people, whose jobs involve the day-to-day use of a database are called as 'Actors on the scene', listed as below.

1.Database Administrators (DBA):

The DBA is responsible for authorizing access to the database, for Coordinating and monitoring its use and for acquiring software and hardware resources as needed. These are the people, who maintain and design the database daily. DBA is responsible for the following issues.

- **Design of the conceptual and physical schemas:**

  The DBA is responsible for interacting with the users of the system to understand what data is to be stored in the DBMS and how it is likely to be used. The DBA creates the original schema by writing a set of definitions and is Permanently stored in the 'Data Dictionary'.

- **Security and Authorization:**

  The DBA is responsible for ensuring the unauthorized data access is not permitted. The granting of different types of authorization allows the DBA to regulate which parts of the database various users can access.

- **Storage structure and Access method definition:**

  The DBA creates appropriate storage structures and access methods by writing a set of definitions, which are translated by the DDL compiler.

- **Data Availability and Recovery from Failures:** The DBA must take steps to ensure that if the system fails, users can continue to access as much of the uncorrupted data as possible. The DBA also work to restore the data to consistent state.

- **Database Tuning:**

  The DBA is responsible for modifying the database to ensure adequate Performance as requirements change. Integrity Constraint Specification: The integrity constraints are kept in a special system structure that is consulted by the DBA whenever an update takes place in the system.

2.Database Designers:

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data.

3. End Users:

People who wish to store and use data in a database.

End users are the people whose jobs require access to the database for querying, updating and generating reports, listed as below.

- **Casual End users:**

  These people occasionally access the database, but they may need different information each time.

- **Naive or Parametric End Users:**

  Their job function revolves around constantly querying and updating the database using standard types of queries and updates.

- **Sophisticated End Users:**

  These include Engineers, Scientists, Business analyst and others familiarize to implement their applications to meet their complex requirements.

- **Stand alone End users:**

These people maintain personal databases by using ready-made program packages that provide easy to use menu based interfaces.

4.System Analyst:

These people determine the requirements of end users and develop specifications for transactions.

5.Application Programmers (Software Engineers):

These people can test, debug, document and maintain the specified transactions.

b. Workers behind the scene:

Database Designers and Implementers:

These people who design and implement the DBMS modules and interfaces as a software package.

2.Tool Developers**:**

Include persons who design and implement tools consisting the packages for design, performance monitoring, and prototyping and test data generation.

3.Operators and maintenance personnel:

These re the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

LEVELS OF DATA ABSTRACTION

This is also called as 'The Three-Schema Architecture', which can be used to separate the user applications and the physical database.

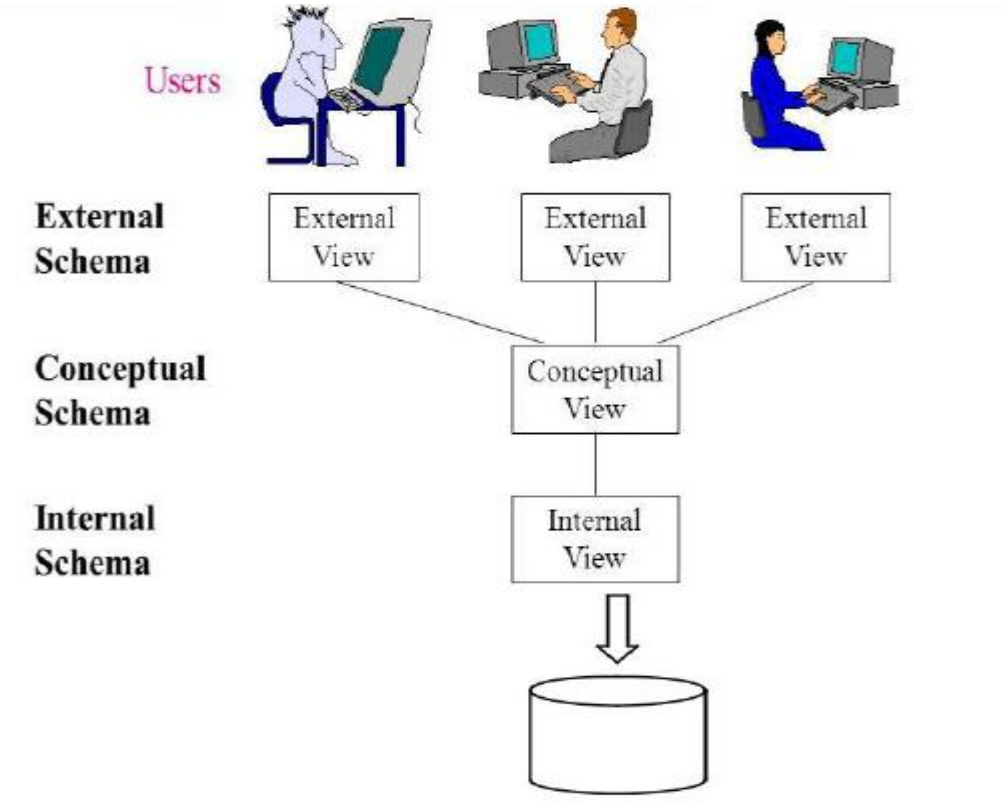


Figure: Levels of Abstraction in a DBMS

1.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..

Example:

Customer account database can be described.

2.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**.

Example:

Each record

type customer = record

cust_name: sting;

cust_city: string;

cust_street: string;

end;

3.Conceptual Level: (or External View / Schema):

The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information 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 their interaction with the system. The system may provide many views for the same database.

Example:

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 consecutive storage locations. 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. 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.

UNIT II

DATABASE DESIGN

ER Model

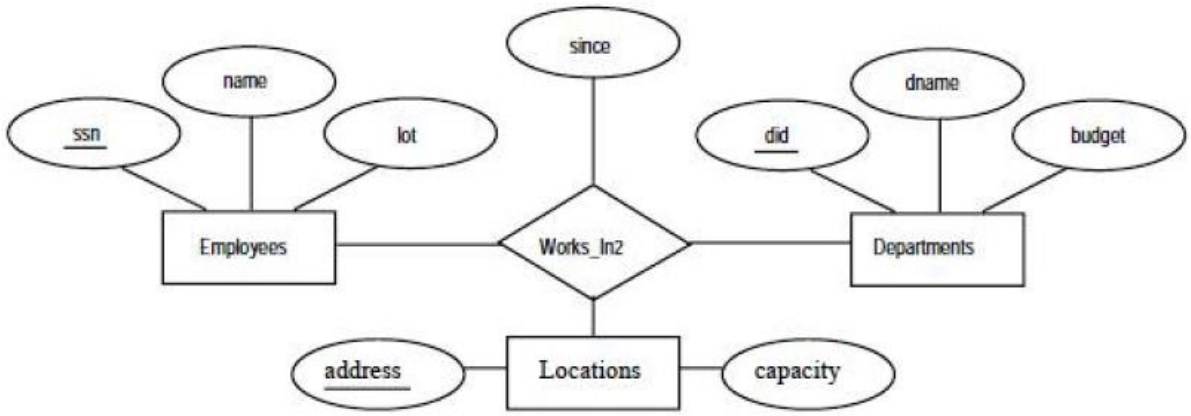
Data Models

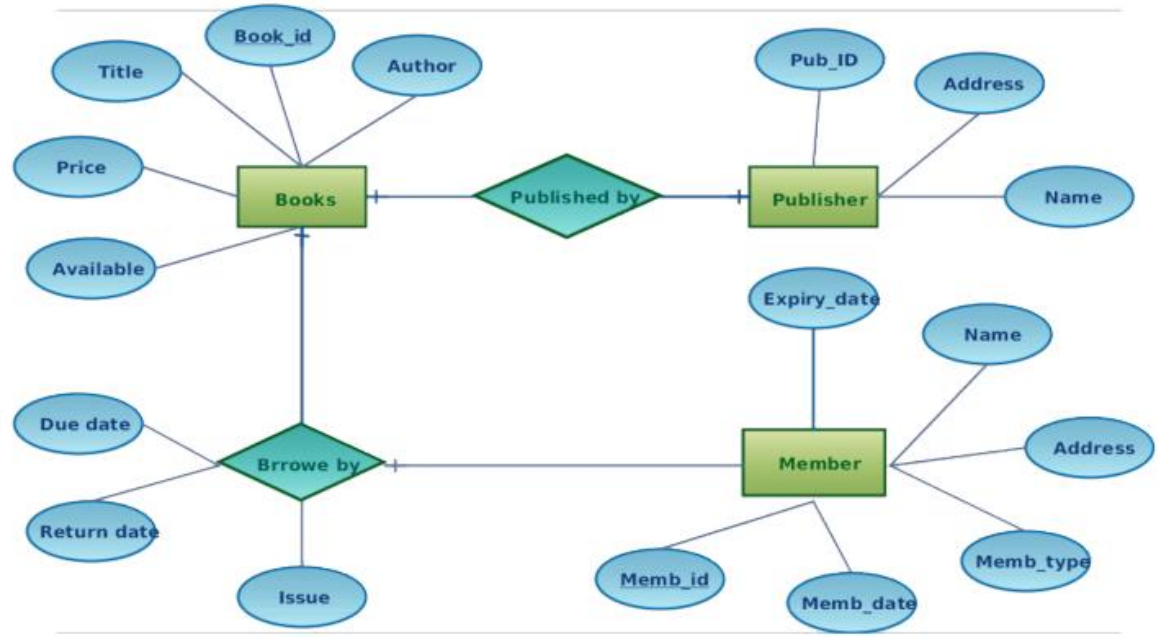
Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

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.
- **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. Suppose that each department has offices in several locations and we want to record the locations at which each employee works. The ER diagram for this variant of Works In, which we call Works In2

Example - ternary





E-R Diagram for Library Management System

E R Model -(Railway Booking System)

E R Model -(Banking Transaction System)

Object-Based Data Model.

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

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 Mark up 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 modelling 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 as organized 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.

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.

• **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.

• **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 we cannot 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..

• **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.

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 labelled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

For example, the data dictionary typically stores descriptions of all:

• Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, data types, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.

• Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.

• Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.

• Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.

• End users and The Administrators of the data base

• Programs that access the database including screen formats, report formats application formats, SQL queries and so on.

• Access authorization for all users of all databases.

• Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database.

Database Users and User Interfaces

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

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*.

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

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

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).

Database Architecture:

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.

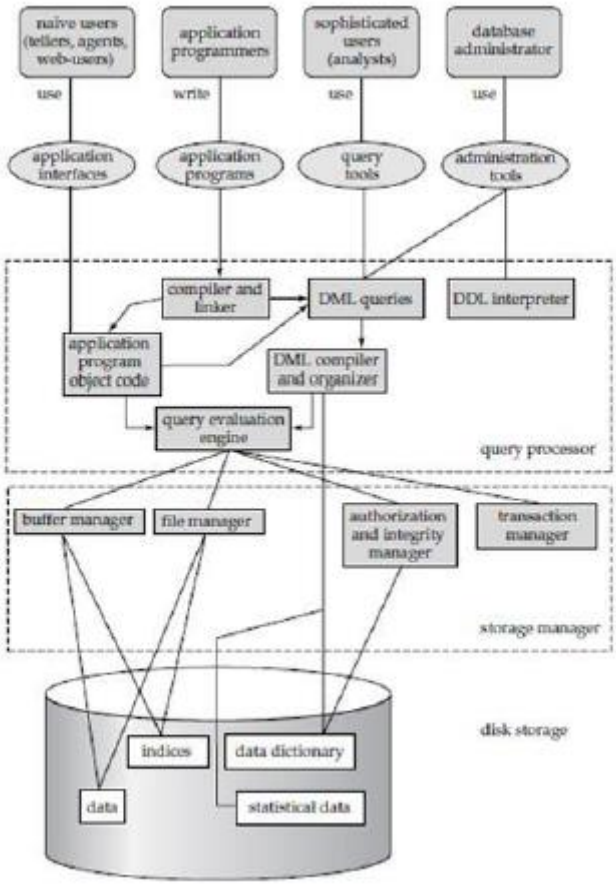


Figure: Database System Architecture

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.

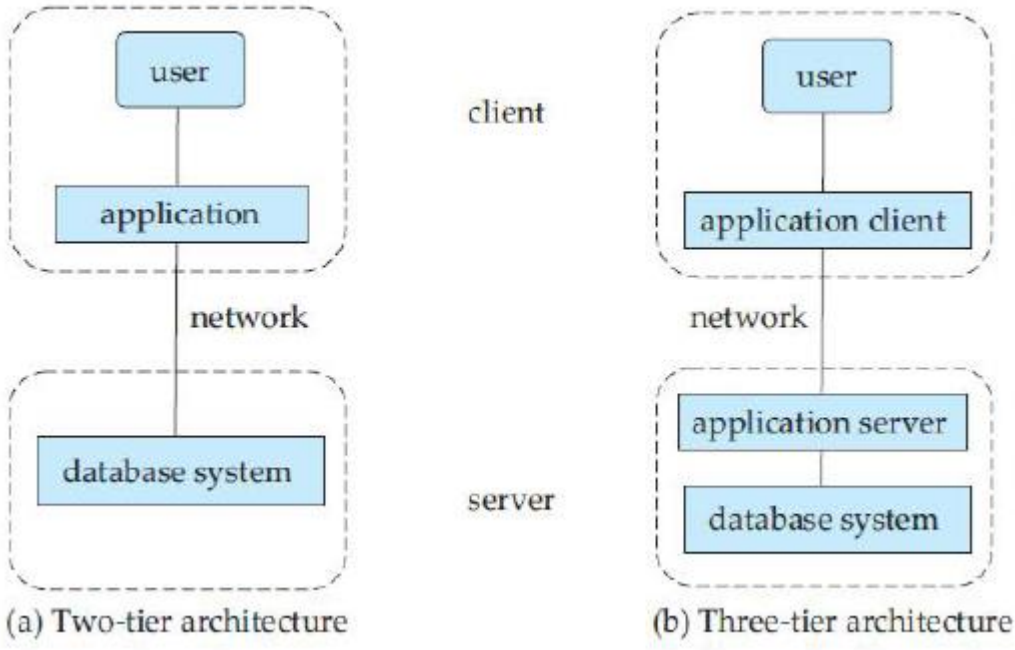


Figure: Two-tier and three-tier architectures.

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 low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The storage manager components include:

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

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

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 violate any database-consistency constraints.

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 canbe 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 attributeconsists 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.

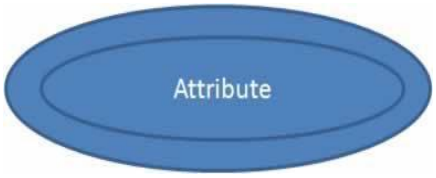
  example for single valued attribute : age of a student. It can take only one value for a particular student.

- **Multi-valued Attributes**

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

  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 attribute

  Example 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.

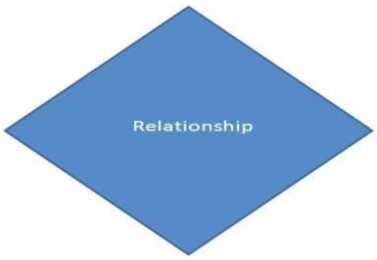
In ER modelling, notation for derived attribute is given below.

Relationships

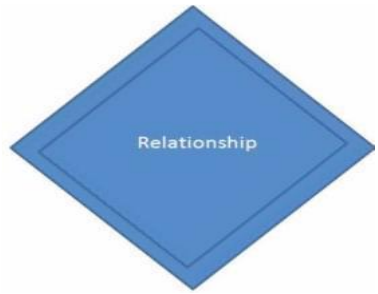
Associations between entities are called relationships

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 employee

Example 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.

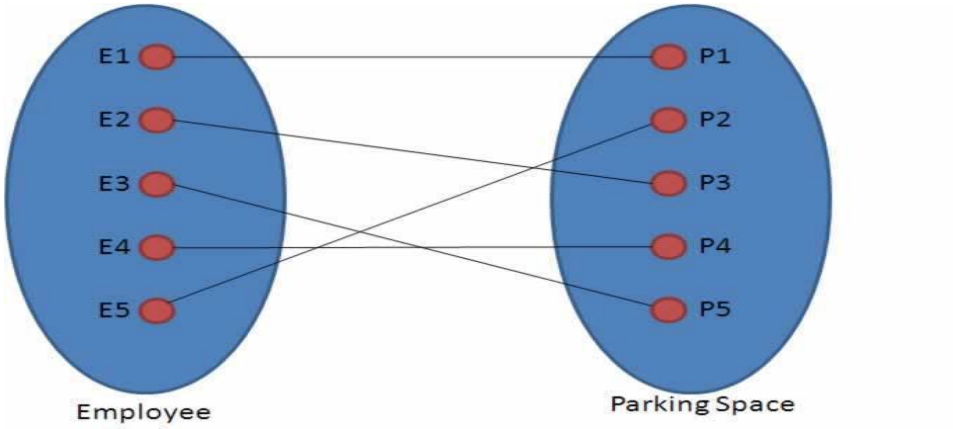
Relationships can have four possible connectivities as given below.

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.

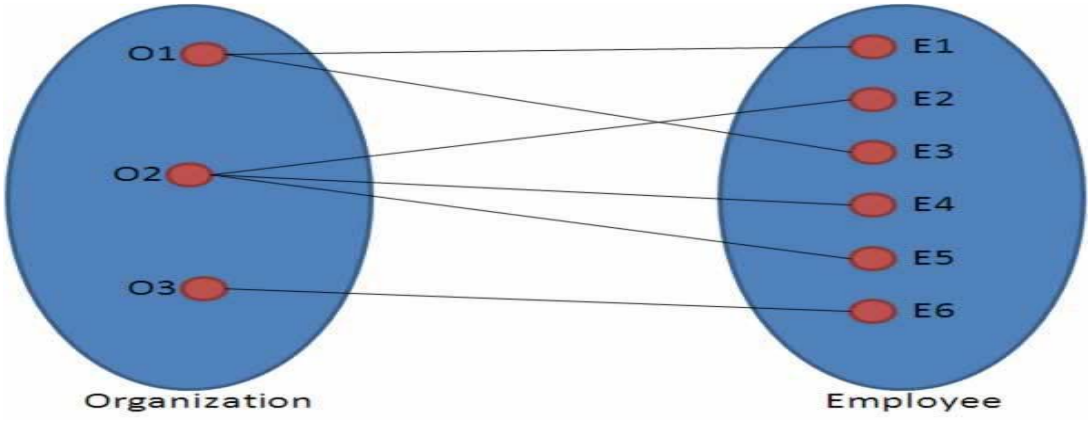


One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

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

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

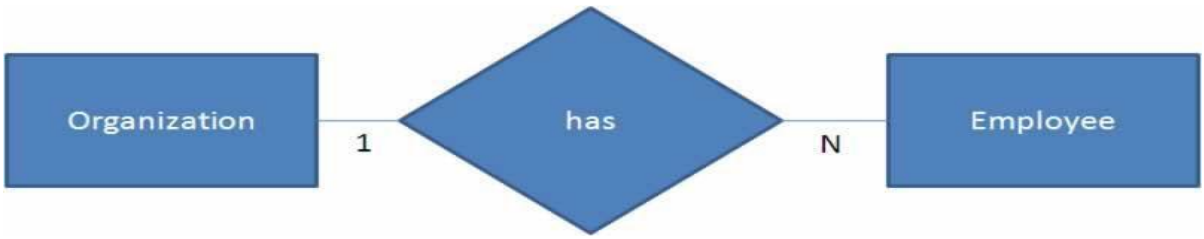
Organization has employees



One organization can have many employees , but one employee works in only one organization.

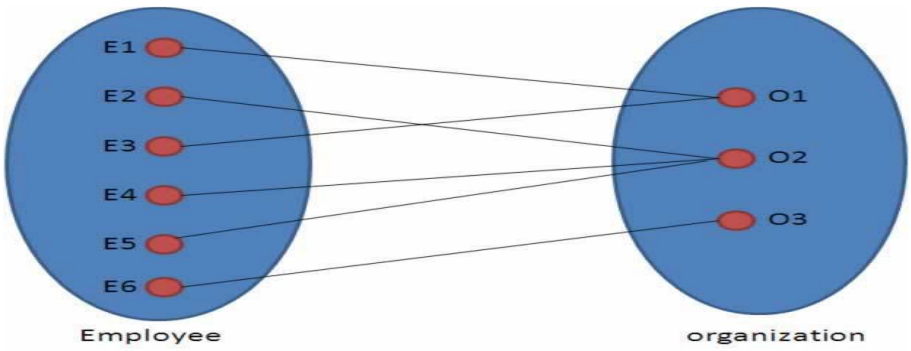
Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

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



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

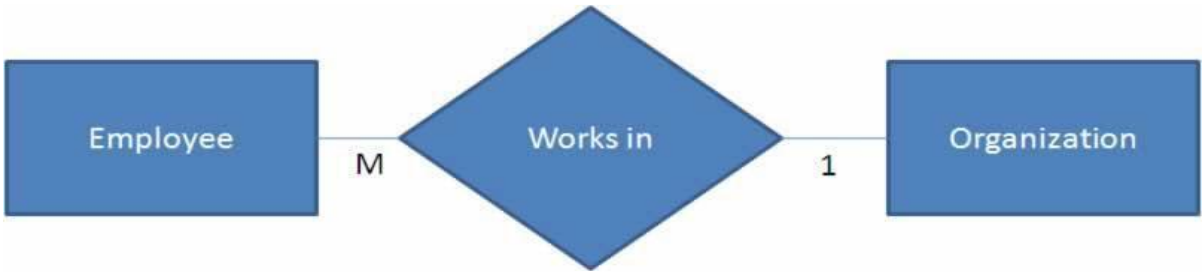
It is the reverse of the One to Many relationship. employee works in organization



One employee works in only one organization But one organization can have many employees.

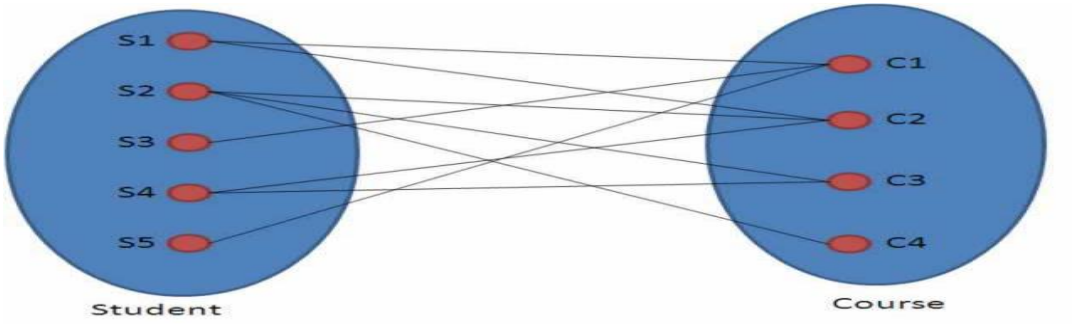
Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

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



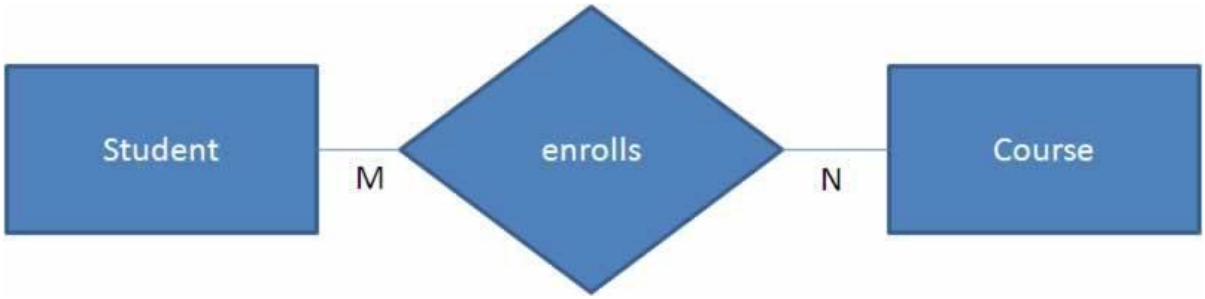
Cardinality – Many-to-Many (M:N)

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.

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.

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*.

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

Keys

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

Example:

The 'students' relation and the constraint that no 2 students have tha same student id (sid).

These can be classified into 3 types as below.

Primary Key:

This is also a candidate key, whose values are used to identify tuples in the relation. It is common to designate one of the candidate keys as a primary key of the relation. The attributes that form the primary key of a relation schema are underlined. It is used to denote a candidate key that is chosen by the database designer as the principal means of identifying entities with an entity set.

A **superkey:**

is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. 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. A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called

candidate keys:

It is 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 *r*1, may include among its attributes the primary key of another relation, say *r*2. This attribute is called a **foreign key** from *r*1, referencing *r*2.

Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**.

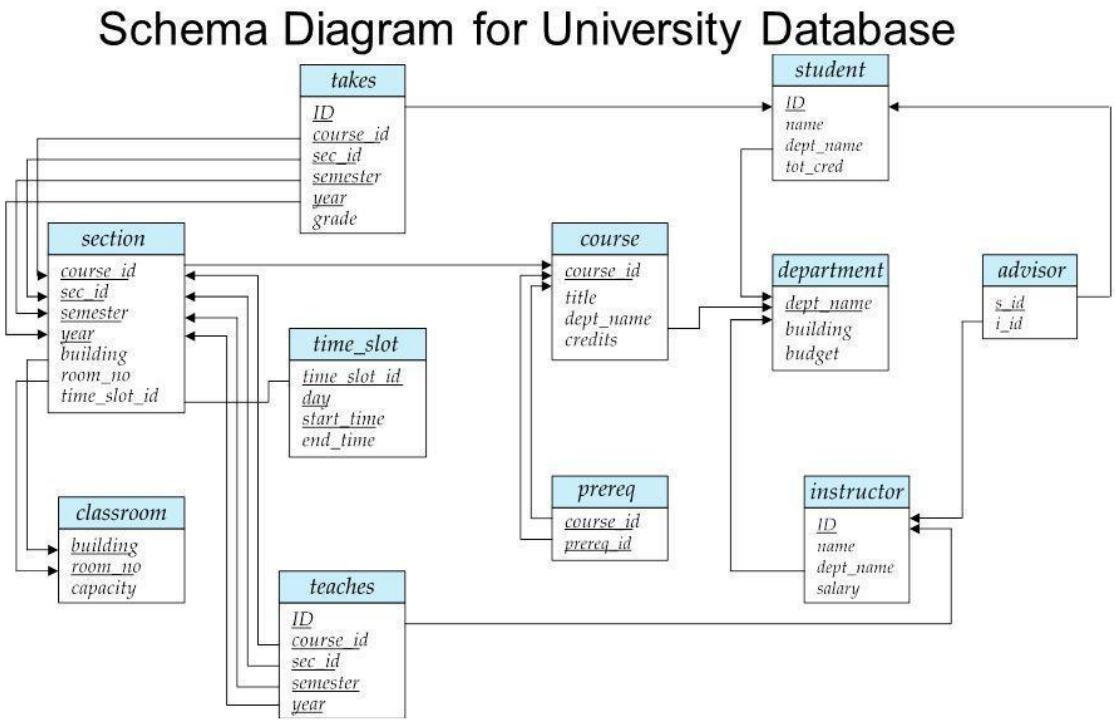


Figure 2.5 : Schema diagram for the university database.

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.

UNIT III

STRUCTURED QUERY LANGUAGE

What is SQL?

- SQL is Structured Query Language, which is a database language designed for the retrieval and management of data in a relational database.
- All the RDBMS systems like MySQL, MS Access, Oracle, Sybase, Postgres, and SQL Server use SQL as their standard database language.

Why to Use SQL?

SQL provides an interface to a relational database.

Here, are important reasons for using SQL

- It helps users to access data in the RDBMS system.
- It helps us to describe the data.
- It allows us to define the data in a database and manipulate that specific data.
- With the help of SQL commands in DBMS, we can create and drop databases and tables.
- SQL offers us to use the function in a database, create a view, and stored procedure.
- We can set permissions on tables, procedures, and views.

History of SQL

"A Relational Model of Data for Large Shared Data Banks" was a paper which was published by the great computer scientist "E.F. Codd" in 1970.

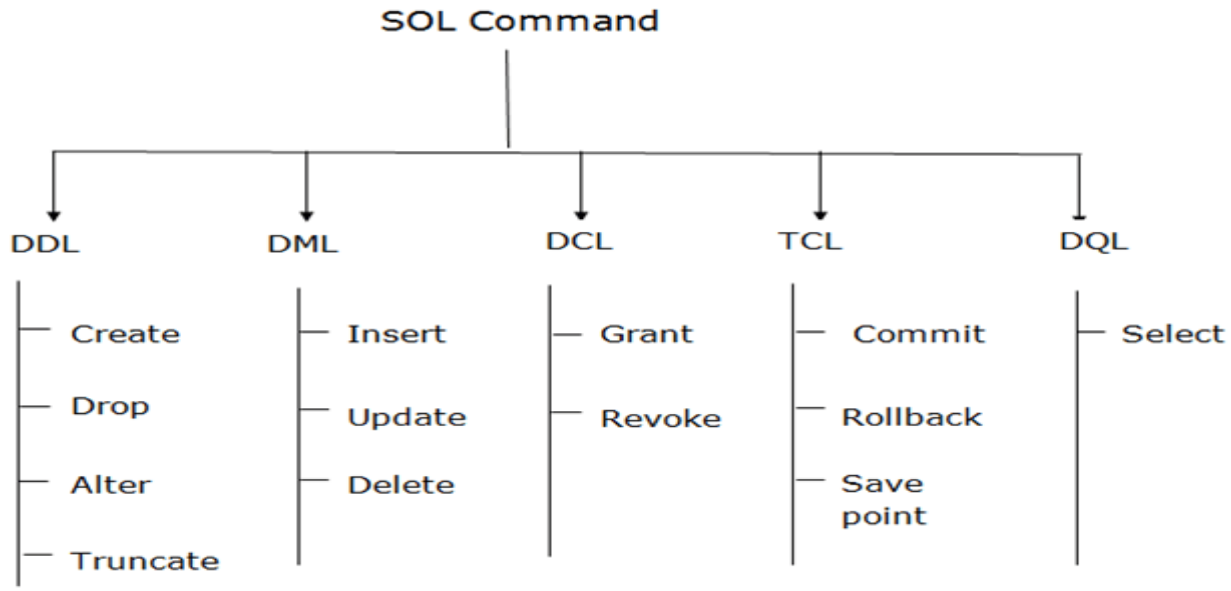
The IBM researchers Raymond Boyce and Donald Chamberlin originally developed the SEQUEL (Structured English Query Language) after learning from the paper given by E.F. Codd. They both developed the SQL at the San Jose Research laboratory of IBM Corporation in 1970. In 1979, Relational Software, Inc. (now Oracle) introduced the first commercially available implementation of SQL.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987.[11] Since then, the standard has been revised to include a larger set of features. Despite the existence of standards, most SQL code requires at least some changes before being ported to different database systems. New versions of the standard were published and most recently, 2016.

Types of SQL

Here are five types of widely used SQL queries.

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language(DCL)
- Transaction Control Language(TCL)
- Data Query Language (DQL)



All operations performed on the information in a database are run using SQL **statements**. A SQL statement consists of identifiers, parameters, variables, names, data types, and SQL **reserved words**.

What is DDL?

Definition: The Language used to define the database structure or schema is called "Data Definition Language".

- The Commands (or) statements used to define the structute of database are:

1. CREATE
2. ALTER
3. DROP
4. TRUNCATE
5. RENAME

1.CREATE

Create command can be used to create

(i) Databases

(ii) Tables and

(iii) Views.

(i)Creating Database

Syntax:

```
create  databasedabasename;
```

Ex: create database MRCET_ITA;

(ii)Creating Table

Syntax:

```
Create table tablename(Columnname1 Datatype,
Columnname2   Datatype,

……..,

Columnnamendatatype);
```

Ex: create table Student(SRno integer(5),

Snamevarchar(20),

Address varchar(15));

2.ALTER Command

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

1. ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_namedatatype;
```

Ex: The following SQL adds an "Email" column to the "Customers" table:

```
ALTERTABLE Customers
        ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

Syntax:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Ex: The following SQL deletes the "Email" column from the "Customers" table:

```
ALTERTABLE Customers
        DROP COLUMN Email;
```

ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

SQL Server / MS Access:

```
ALTER TABLE table_name
ALTER COLUMN column_namedatatype;
```

```
Ex: ALTER TABLE supplier
    ALTER COLUMN supplier_nameVARCHAR(100) NOT NULL;
```

My SQL / Oracle (prior version 10G):

```
ALTER TABLE table_name
MODIFY COLUMN column_namedatatype;
```

Example 1: Modifying single Column

```
ALTER TABLE supplier
 MODIFY supplier_namechar(100) NOT NULL;
```

Example 2: Modifying Multiple Columns

```
ALTER TABLE supplier
 MODIFY supplier_nameVARCHAR(100) NOT NULL,
 MODIFY city VARCHAR(75);
```

Oracle 10G and later:

```
ALTER TABLE table_name
MODIFY column_namedatatype;
```

3.Drop Command

Syntax

To drop a column in an existing table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
  DROP COLUMN column_name;
```

Example

Let's look at an example that drops (ie: deletes) a column from a table.

For example:

```
ALTER TABLE supplier
  DROP COLUMN supplier_name;
```

This SQL ALTER TABLE example will drop the column called ***supplier_name*** from the table called ***supplier***.

TRUNCATE:

This command used to delete all the rows from the table and free the space containing the table.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE table students;
```

What is Data Manipulation Language?

Data Manipulation Language (DML) allows user to modify the database instance by inserting, modifying, and deleting its data. It is responsible for performing all types data modification in a database.

There are three basic constructs which allow database program and user to enter data and information are:

Here are some important DML commands in SQL:

- INSERT
- UPDATE
- DELETE

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

Syntax:

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

For example:

```
INSERT INTO students (RollNo, FIrstName, LastName) VALUES ('60', 'Tom', Erichsen');
```

UPDATE:

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

Syntax:

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

For example:

```
UPDATE students
SET FirstName = 'Jhon', LastName= 'Wick'
WHERE StudID = 3;
```

DELETE:

This command is used to remove one or more rows from a table.

Syntax:

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

For example:

```
DELETE FROM students
WHERE FirstName = 'Jhon';
```

What is DCL?

DCL (Data Control Language) includes commands like GRANT and REVOKE, which are useful to give "rights & permissions." Other permission controls parameters of the database system.

Examples of DCL commands:

Commands that come under DCL:

- Grant
- Revoke

Grant:

This command is use to give user access privileges to a database.

Syntax:

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

For example:

```
GRANT SELECT ON Users TO'Tom'@'localhost;
```

Revoke:

It is useful to back permissions from the user.

Syntax:

```
REVOKE privilege_nameONobject_nameFROM {user_name |PUBLIC |role_name}
```

For example:

```
REVOKE SELECT, UPDATE ON student FROM BCA, MCA;
```

What is TCL?

Transaction control language or TCL commands deal with the transaction within the database.

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

Syntax:

```
Commit;
```

For example:

```
DELETE FROM Students
WHERE RollNo =25;
COMMIT;
```

Rollback

Rollback command allows you to undo transactions that have not already been saved to the database.

Syntax:

```
ROLLBACK;
```

Example:

```
DELETE FROM Students
WHERE RollNo =25;
```

SAVEPOINT

This command helps you to sets a savepoint within a transaction.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;
```

Example:

```
SAVEPOINT RollNo;
```

What is DQL?

Data Query Language (DQL) is used to fetch the data from the database. It uses only one command:

SELECT:

This command helps you to select the attribute based on the condition described by the WHERE clause.

Syntax:

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

For example:

```
SELECT FirstName
FROM Student
```

```
WHERE RollNo> 15;
```

TCL Commands

TCL Commands in SQL- Transaction Control Language Examples: Transaction Control Language can be defined as the portion of a database language used for maintaining consistency of the database and managing transactions in database. A set of SQL statements that are co-related logically and executed on the data stored in the table is known as transaction. In this tutorial, you will learn different TCL Commands in SQL with examples and differences between them.

1. Commit Command
2. Rollback Command
3. Savepoint Command

TCL Commands in SQL- Transaction Control Language Examples

The modifications made by the DML commands are managed by using TCL commands. Additionally, it makes the statements to grouped together into logical transactions.

TCL Commands

There are three commands that come under the TCL:

1. **Commit**

The main use of Commit command is to make the transaction permanent. If there is a need for any transaction to be done in the database that transaction permanent through commit command.

Syntax:

COMMIT;

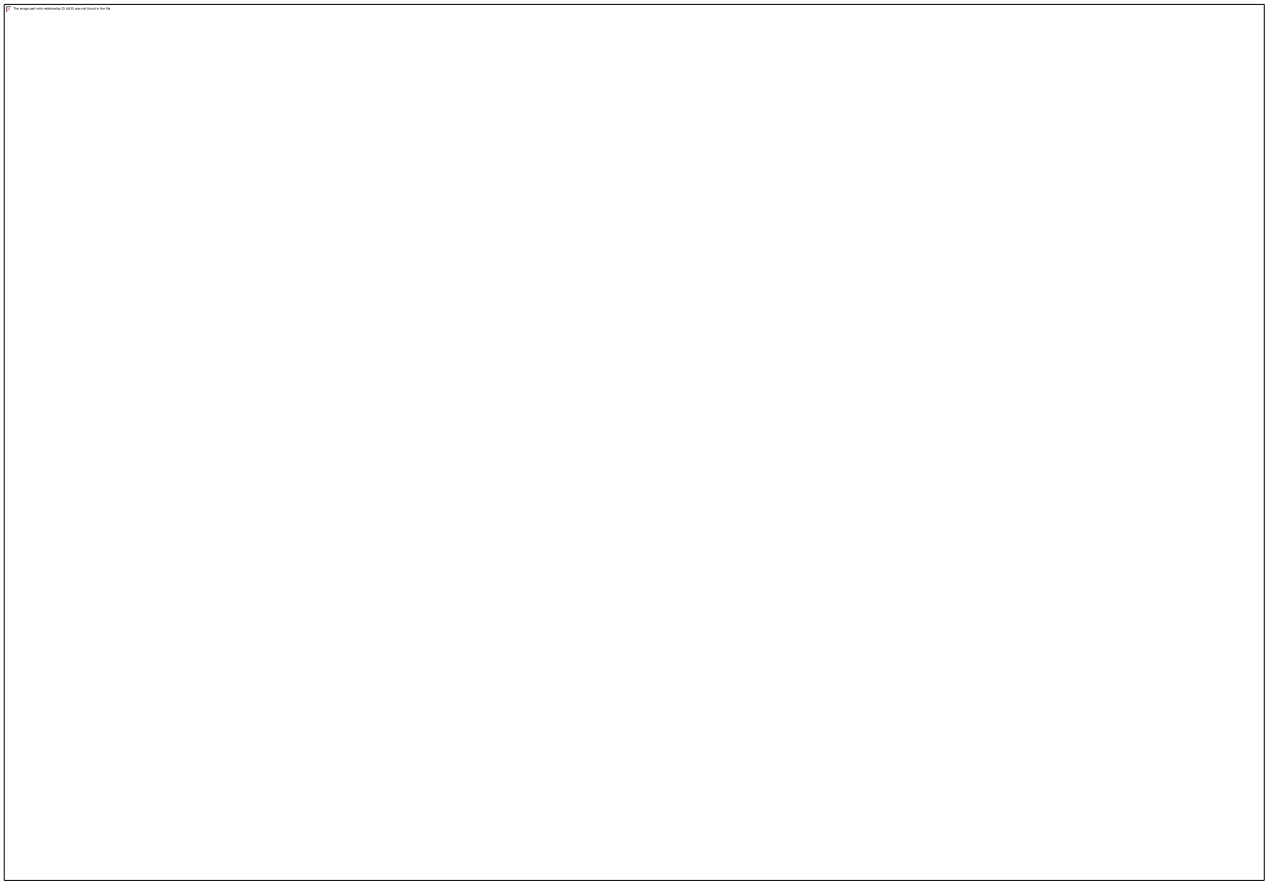
For Example

UPDATE STUDENT SET STUDENT_NAME = 'Maria' WHERE STUDENT_NAME = 'Meena';

COMMIT;

- By using the above set of instructions, you can update the wrong student name by the correct one and save it permanently in the database. The update transaction gets completed when commit is used. If commit is not used, then there will be lock on 'Meena' record till the rollback or commit is issued.

- Now have a look at the below diagram where ‘Meena’ is updated and there is a lock on her record. The updated value is permanently saved in the database after the use of commit and lock is released.



2. **Rollback**

- Using this command, the database can be restored to the last committed state.
- Additionally, it is also used with savepoint command for jumping to a savepoint in a transaction.

Syntax:

Rollback to savepoint-name;

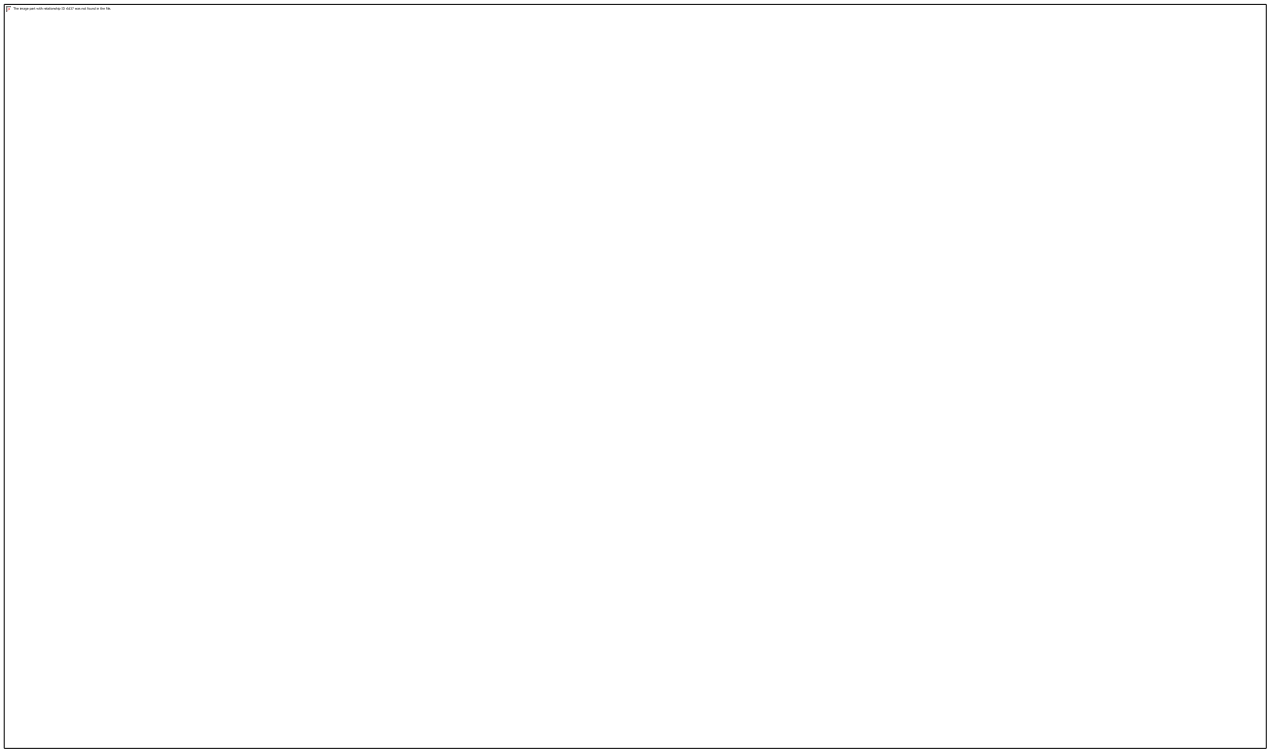
For example

UPDATE STUDENT SET STUDENT_NAME = ‘Manish’ WHERE STUDENT_NAME = ‘Meena’; ROLLBACK;

- This command is used when the user realizes that he/she has updated the wrong information after the student name and wants to undo this update.

- The users can issues ROLLBACK command and then undo the update.

Have a look at the below tables to know better about the implementation of this command.



3. **Savepoint**

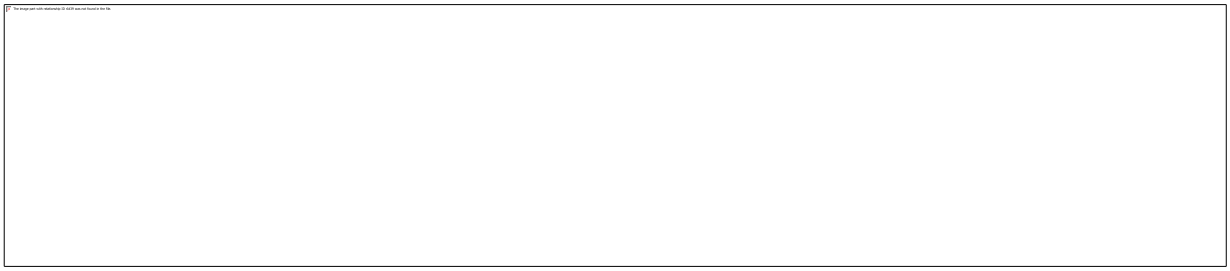
The main use of the Savepoint command is to save a transaction temporarily. This way users can rollback to the point whenever it is needed.

The general syntax for the savepoint command is mentioned below:

savepointsavepoint-name;

For Example

Following is the table of a school class



Use some SQL queries on the above table and then watch the results

INSERT into CLASS VALUES (101, ‘Rahul);

Commit;

UPDATE CLASS SET NAME= ‘Tyler’ where id= 101;

SAVEPOINT A;

INSERT INTO CLASS VALUES (102, ‘Zack’);

Savepoint B;

INSERT INTO CLASS VALUES (103, ‘Bruno’)

Savepoint C;

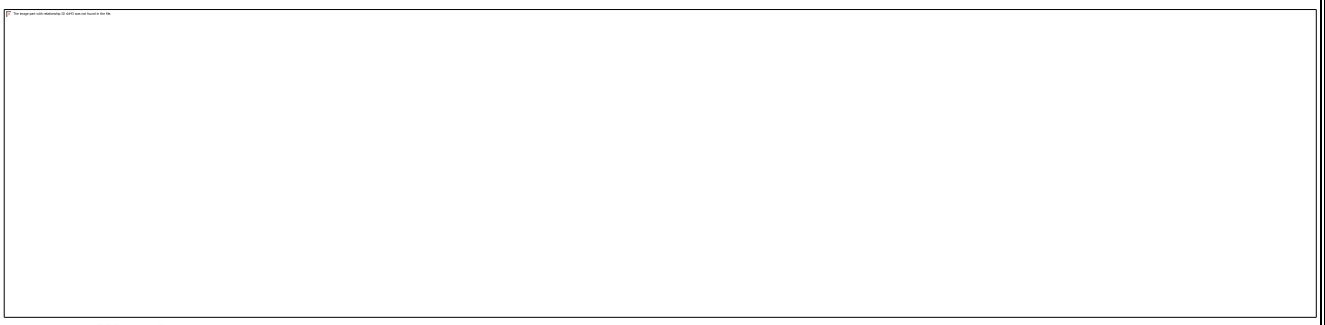
Select * from Class;

The result will look like

Now rollback to savepoint B

Rollback to B;

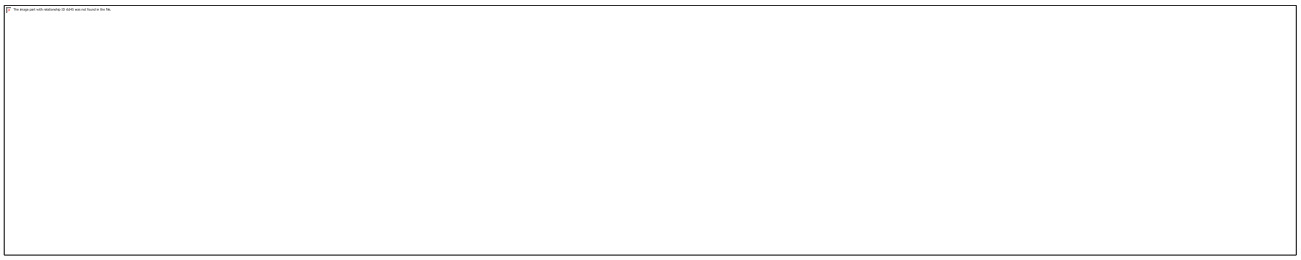
SELECT * from Class;



Now rollback to savepoint A

rollback to A;

SELECT * from class;



Difference between rollback, commit and savepointtcl commands in SQL.

	Rollback	Commit	Savepoint

1.	Database can be restored to the last committed state	Saves modification made by DML Commands and it permanently saves the transaction.	It saves the transaction temporarily.
2.	Syntax- ROLLBACK [To SAVEPOINT_NAME];	Syntax- COMMIT;	Syntax- SAVEPOINT [savepoint_name;]
3.	Example- ROLLBACK Insert3;	Example- SQL> COMMIT;	Example- SAVEPOINT table_create;

```
START TRANSACTION;
savepoint a;
update t1 set n1=18 where n1=13;
rollbackto a;
```

In relational database the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called **STUDENT** which stores **ROLL_NO**, **NAME**, **ADDRESS**, **PHONE** and **AGE** of students.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

These are some important terminologies that are used in terms of relation.

Attribute: Attributes are the properties that define a relation. e.g.; **ROLL_NO, NAME** etc.

Tuple: Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1 RAM DELHI 9455123451 18

Degree: The number of attributes in the relation is known as degree of the relation. The STUDENT relation defined above has degree 5.

Cardinality: The number of tuples in a relation is known as cardinality. The STUDENT relation defined above has cardinality 4.

Column: Column represents the set of values for a particular attribute. The column ROLL_NO isextracted from relation STUDENT.

ROLL_NO

1

2

3

4

SQL Set Operations

Set operations allow the results of multiple queries to be combined into a single result set.

The **Set Operators** combine a similar type of data from two or more SQL database tables. It mixes the result, which is extracted from two or more SQL queries, into a single result.

Set operators combine more than one select statement in a single query and return a specific result set.

Set operators include `UNION`, `INTERSECT`, and `EXCEPT`.

UNION

In SQL the UNION clause combines the results of two SQL queries into a single table of all matching rows. The two queries must result in the same number of columns and compatible data types in order to unite. Any duplicate records are automatically removed unless UNION ALL is used.

Syntax of UNION:

SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]

UNION

SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];

A simple example would be a database having tables sales2005 and sales2006 that have identical structures but are separated because of performance considerations. A UNION query could combine results from both tables.

Note that UNION ALL does not guarantee the order of rows. Rows from the second operand may appear before, after, or mixed with rows from the first operand. In situations where a specific order is desired, ORDER BY must be used.

Note that UNION ALL may be much faster than plain UNION.

sales2005	
person	**amount**
Joe	1000
Alex	2000

Bob	5000

sales2006

person	amount
Joe	2000
Alex	2000
Zach	35000

Executing this statement:

SELECT * FROM sales2005UNIONSELECT * FROM sales2006;

yields this result set, though the order of the rows can vary because no ORDER BY clause was supplied:

person	amount
Joe	1000
Alex	2000
Bob	5000

Joe	2000
Zach	35000

UNION ALL gives different results, because it will not eliminate duplicates. Executing this statement:

SELECT * FROM sales2005UNION ALLSELECT * FROM sales2006;

would give these results, again allowing variance for the lack of an ORDER BY statement:

person	amount
Joe	1000
Joe	2000
Alex	2000
Alex	2000
Bob	5000
Zach	35000

INTERSECT

The SQL INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets. For purposes of duplicate removal the INTERSECT operator does not distinguish between NULLs.

The INTERSECT operator removes duplicate rows from the final result set. The INTERSECT ALL operator does not remove duplicate rows from the final result set, but if a row appears X times in the first query and Y times in the second, it will appear min(X, Y) times in the result set.

The data type and the number of columns must be the same for each SELECT statement used with the INTERSECT operator.

Syntax of INTERSECT

SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]

INTERSECT

SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Employee_details1:

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Employee_details2:

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Suppose, we want to see a common record of the employee from both the tables in a single output. For this, we have to write the following query in SQL:

SELECT Emp_Name FROM Employee_details1

INTERSECT

SELECT Emp_Name FROM Employee_details2 ;

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
203	Saket	30000	Aligarh

EXCEPT

The SQL EXCEPT operator takes the distinct rows of one query and returns the rows that do not appear in a second result set. For purposes of row elimination and duplicate removal, the EXCEPT operator does not distinguish between NULLs. The EXCEPT ALL operator does not remove duplicates, but if a row appears X times in the first query and Y times in the second, it will appear max(X - Y, 0) times in the result set.

Notably, the Oracle platform provides a MINUS operator which is functionally equivalent to the SQL standard EXCEPT DISTINCT operator.

The following example EXCEPT query returns all rows from the Orders table where Quantity is between 1 and 49, and those with a Quantity between 76 and 100.

Worded another way; the query returns all rows where the Quantity is between 1 and 100, apart from rows where the quantity is between 50 and 75.

```
SELECT *FROM   Orders WHERE Quantity BETWEEN 1 AND 100

EXCEPT

SELECT *FROM   Orders WHERE Quantity BETWEEN 50 AND 75;
```

Joins

A join is a query that combines rows from two or more tables, views, based on a common field between them.

Consider the following two tables −

Table 1 − CUSTOMERS Table

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP        |4500.00|
|7|Muffy|24|Indore|10000.00|
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
|102|2009-10-0800:00:00|3|3000|
```

```
|100|2009-10-0800:00:00|3|1500|
|101|2009-11-2000:00:00|2|1560|
|103|2008-05-2000:00:00|4|2060|
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SELECT ID, NAME, AGE,AMOUNT  FROM CUSTOMERS, ORDERS
WHERE  CUSTOMERS.ID= ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

SQL JOINS: EQUI JOIN and NON EQUI JOIN

The are two types of SQL JOINS - EQUI JOIN and NON EQUI JOIN

1) SQL EQUI JOIN:

The SQL EQUI JOIN is a simple SQL join uses the equal sign(=) as the comparison operator for the condition. It has two types - SQL Outer join and SQL Inner join.

2) SQL NON EQUI JOIN :

The SQL NON EQUI JOIN is a join uses comparison operator other than the equal sign like >, <, >=, <= with the condition.

SQL EQUI JOIN : INNER JOIN and OUTER JOIN

The SQL EQUI JOIN can be classified into two types - **INNER JOIN** and **OUTER JOIN**

1. **SQL INNER JOIN**

This type of EQUI JOIN returns all rows from tables where the key record of one table is equal to the key records of another table.

2. **SQL OUTER JOIN**

This type of EQUI JOIN returns all rows from one table and only those rows from the secondary table where the joined condition is satisfying i.e. the columns are equal in both tables.

In order to perform a JOIN query, the required information we need are:

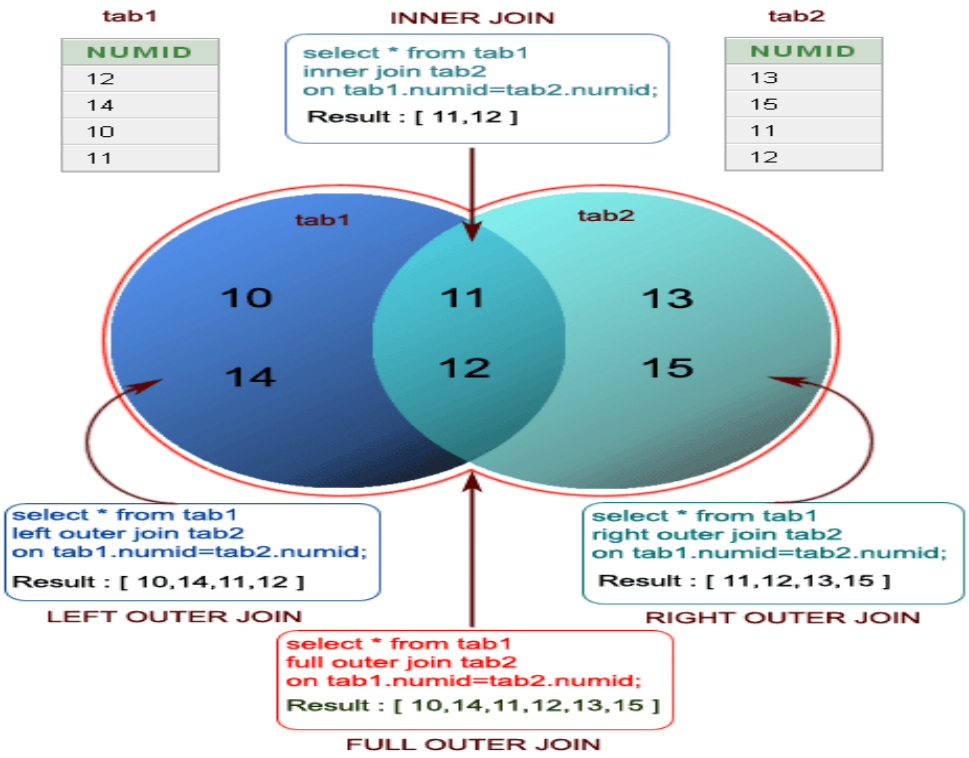
a) The name of the tables

b) Name of the columns of two or more tables, based on which a condition will perform.

Syntax:

FROM table1
join_type table2
[ON (join_condition)]
ON can be replaced with WHERE

Pictorial Presentation of SQL Joins:



Let's Consider the two tables given below.

Table name- Student:

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

Table name — Record:

id	Class	City
9	3	Delhi
10	2	Delhi
12	2	Delhi

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;
```

Output :

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

name	Id	class	City
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

name	Id	city
Gouri	10	Delhi
Hina	12	Delhi
Megha	12	Delhi
Gouri	12	Delhi

Nested Queries in SQL:

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. Nested Queries are also called as**subqueries**.

Subqueries are useful when you must execute multiple queries to solve a single problem. Each query portion of a statement is called a query block. In the following query, the subquery in parentheses is the inner query block:

```
SELECT first_name, last_name FROM   employees
WHERE  department_id
IN     ( SELECTdepartment_id
         FROM departments
         WHERE location_id = 1800 );
```

- The inner SELECT statement retrieves the IDs of departments with location ID 1800. These department IDs are needed by the outer query block, which retrieves names of employees in the departments whose IDs were supplied by the subquery.
- The structure of the SQL statement does not force the database to execute the inner query first. For example, the database could rewrite the entire query as a join of employees and departments, so that the subquery never executes by itself.

Subqueries can be **correlated** or **uncorrelated**.

Correlated subquery - In correlated subquery, inner query is dependent on the outer query. Outer query needs to be executed before inner query

Non-Correlated subquery - In non-correlated query inner query does not dependent on the outer query. They both can run separately.

Correlated Subqueries

A correlated subquery typically obtains values from its outer query before it executes. When the subquery returns, it passes its results to the outer query.

In the following example, the subquery needs values from the **addresses.state** column in the outer query:

```
=> SELECT name, street, city, state FROM addresses
    WHERE EXISTS (SELECT * FROM states WHERE states.state = addresses.state);
```

This query is executed as follows:

- The query extracts and evaluates each addresses.state value in the outer subquery records.
- Then the query—using the EXISTS predicate—checks the addresses in the inner (correlated) subquery.
- Because it uses the EXISTS predicate, the query stops processing when it finds the first match.

NoncorrelatedSubqueries

A noncorrelatedsubquery executes independently of the outer query. The subquery executes first, and then passes its results to the outer query, For example:

```
=> SELECT name, street, city, state FROM addresses WHERE state IN (SELECT state
FROM states);
```

This query is executed as follows:

- Executes the subquery SELECT state FROM states (in bold).
- Passes the subquery results to the outer query.

A query's WHERE and HAVING clauses can specify noncorrelatedsubqueries if the subquery resolves to a single row, as shown below:

In WHERE clause

=> SELECT COUNT(*) FROM SubQ1 WHERE SubQ1.a = (SELECT y from SubQ2);

In HAVING clause

=> SELECT COUNT(*) FROM SubQ1 GROUP BY SubQ1.a HAVING SubQ1.a = (SubQ1.a & (SELECT y from SubQ2)

Aggregate functions:

Aggregate functions operate on values across rows to perform mathematical calculations such as sum, average, counting, minimum/maximum values, standard deviation, and estimation, as well as some non-mathematical operations.

An aggregate function takes multiple rows (actually, zero, one, or more rows) as input and produces a single output.

Various Aggregate Functions:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Let us consider a table that contains the following data:

```
selectx,yfromsimpleorderbyx,y;
+----+----+
|  X |  Y |
|----+----|
| 10 | 20 |
| 20 | 44 |
| 30 | 70 |
+----+----+
```

The aggregate function returns one output row for multiple input rows:

```
selectsum(x)
```

```
fromsimple;
+--------+
| SUM(X) |
|--------|
|     60 |
+--------+
```

Now let us understand each Aggregate function with a example:

Id Name Salary

---

1 A 80

2 B 40

3 C 60

4 D 70

5 E 60

6 F Null

Count():

Count(*): Returns total number of records .i.e 6.

Count(salary): Return number of Non Null values over the column salary. i.e 5.

Count(Distinct Salary): Return number of distinct Non Null values over the column salary .i.e 4.

Sum():

sum(salary): Sum all Non Null values of Column salary i.e., 310

sum(Distinct salary): Sum of all distinct Non-Null values i.e., 250.

Avg():

Avg(salary) = Sum(salary) / count(salary) = 310/5

Avg(Distinct salary) = sum(Distinct salary) / Count(Distinct Salary) = 250/4

Min():

Min(salary): Minimum value in the salary column except NULL i.e., 40.

Max(salary): Maximum value in the salary i.e., 80.

Aggregate Functions and NULL Values

Some aggregate functions ignore NULL values. For example, AVG calculates the average of values 1, 5, and NULL to be 3, based on the following formula:

(1 + 5) / 2 = 3

If all of the values passed to the aggregate function are NULL, then the aggregate function returns NULL.

Some aggregate functions can be passed more than one column. For example:

select count(col1, col2) from table1;

In these instances, the aggregate function ignores a row if any individual column is NULL.

```
insertintot(x,y)values
(1,2),-- No NULLs.
(3,null),-- One but not all columns are NULL.
(null,6),-- One but not all columns are NULL.
(null,null);-- All columns are NULL.
```

Query the table:

```
selectcount(x,y)fromt;
+-------------+
| COUNT(X, Y) |
|-------------|
|           1 |
+-------------+
```

Similarly, if SUM is called with an expression that references two or more columns, and if one or more of those columns is NULL, then the expression evaluates to NULL, and the row is ignored:

```
selectsum(x+y)fromt;
+------------+
```

```
| SUM(X + Y) |
|------------|
|          3 |
+------------+
```

SQL also provides a special comparison operator IS NULL to test whether a column value is null; for example the value of y IS NULL returns **true** when x is 3 and IS NOT NULL returns **false**.

INTRODUCTION TO VIEWS

A view is a table whose rows are not explicitly stored in the database but are computed as needed.

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

Sample Tables:

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATING VIEWS

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

Syntax:

CREATE VIEW view_name ASSELECT column1, column2.....
FROM table_nameWHERE condition;

view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows

Examples:

Creating View from a single table:
In this example we will create a View named DetailsView from the table StudentDetails.
Query:

CREATE VIEW DetailsView ASSELECT NAME, ADDRESS
FROM StudentDetailsWHERE S_ID < 5;

To see the data in the View, we can query the view in the same manner as we query a table.

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Creating View from multiple tables: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

Query:

CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;

To display data of View MarksView:

SELECT * FROM MarksView;

Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

DELETING VIEWS

SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

DROP VIEW view_name;

view_name: Name of the View which we want to delete.
For example, if we want to delete the View MarksView, we can do this as:

DROP VIEW MarksView;

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.

5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.

Syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

For example, if we want to update the view MarksView and add the field AGE to this View from StudentMarks Table, we can do this as:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS,
StudentMarks.AGEFROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

Inserting a row in a view:

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.**Syntax**:

```
INSERT INTO view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);
```

view_name: Name of the View

Example:

In the below example we will insert a new row in the View DetailsView which we have created above in the example of “creating views from a single table”.

INSERT INTO DetailsView(NAME, ADDRESS)

VALUES("Suresh","Gurgaon");

If we fetch all the data from DetailsView now as,

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

Deleting a row from a View:

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.**Syntax**:

DELETE FROM view_name

WHERE condition;

view_name:Name of view from where we want to delete rows

condition: Condition to select rows

Example:

In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

DELETE FROM DetailsView

WHERE NAME="Suresh";

If we fetch all the data from DetailsView now as,

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

TRIGGERS

A trigger is a stored procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

Event: A change to the database that activates the trigger.

Condition: A query or test that is run when the trigger is activated.

Action: A procedure that is executed when the trigger is activated and its con-dition is true.

A **trigger action** can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database.

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.

Examples of Triggers in SQL

The trigger called init count initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called incr count increments the counter for each inserted tuple that satisfies the condition age < 18.

```
CREATE TRIGGER init count BEFORE INSERT ON Students /* Event */

DECLARE

count INTEGER;

BEGIN

count := 0;

END

/* Action */

CREATE TRIGGER incr count AFTER INSERT ON Students /* Event */

WHEN (new.age< 18) /* Condition; ‘new’ is just-inserted tuple */

FOR EACH ROW

BEGIN /* Action; a procedure in Oracle’s PL/SQL syntax */
```

count := count + 1;

END

(identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with age < 18. (The trigger in Figure 5.19 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

CREATE TRIGGER set count AFTER INSERT ON Students /* Event */

REFERENCING NEW TABLE AS InsertedTuples

FOR EACH STATEMENT

INSERT /* Action */

INTO StatisticsTable(ModifiedTable, ModificationType, Count) SELECT

‘Students’, ‘Insert’, COUNT * FROM InsertedTuples I WHERE I.age< 18

UNIT IV

DEPENDENCIES AND NORMAL FORMS

Importance of a good schema design

What is a Database Schema?

A database schema is a blueprint that represents the tables and relations of a data set. Good database schema design is essential to making your data tractable so that you can make sense of it and build the dashboards, reports, and data models that you need.

It is important to have a good database schema design. The reasons are:

- Without a good database design, the database is likely to be unsatisfactory.
- A good database design must be implemented in such ways that the queries are written in a simple and easier manner.
- A good database design doesn't have data redundancies (data redundancy refers to duplication of data.).
- The accuracy must be good enough after the implementation of good database design.

Four specific issues resulting from bad schema design:

1. **Referential Integrity**: a poorly done database design leaves the application vulnerable to referential integrity issues.
2. **Scalability:** a poorly done design would struggle to scale when future application functionality is added.
3. **Performance**: over- or under-normalization can result in significant performance issues in the application that attempts to work with the model.
4. **Maintainability**: a poor database design will make life miserable for developers attempting to code against the model or to comprehend the model to diagnose issues.

Introduction of Database Normalization

Database normalization is the process of structuring and handling the relationship between data to minimize redundancy in the relational table and avoid the unnecessary anomalies properties from the database like insertion, update and delete. It helps to divide large database

tables into smaller tables and make a relationship between them. It can remove the redundant data and ease to add, manipulate or delete table fields.

A normalization defines rules for the relational table as to whether it satisfies the normal form. A **normal form** is a process that evaluates each relation against defined criteria and removes the multi valued, joins, functional and trivial dependency from a relation. If any data is updated, deleted or inserted, it does not cause any problem for database tables and help to improve the relational table' integrity and efficiency.

Objective of Normalization

1. It is used to remove the duplicate data and database anomalies from the relational table.
2. Normalization helps to reduce redundancy and complexity by examining new data types used in the table.
3. It is helpful to divide the large database table into smaller tables and link them using relationship.
4. It avoids duplicate data or no repeating groups into a table.
5. It reduces the chances for anomalies to occur in a database.

Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is indicated by the representation below :

$X \rightarrow Y$

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

For example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.

A **function dependency**A → B means for all instances of a particular value of A, there is the same value of B.

For example in the below table A → B is true, but B → A is not true as there are different values of A for B = 3.

A B

---

1 3

2 3

4 0

1 3

4 0

Trivial Functional Dependency

- A → B has trivial functional dependency if B is a subset of A.
- The following dependencies are also trivial like: A → A, B → B

Examples

- ABC → AB
- ABC → A
- ABC → ABC

Non Trivial Functional Dependencies

X → Y is a non trivial functional dependency when Y is not a subset of X.

- X → Y is called completely non-trivial when X intersect Y is NULL.

Example:

- Id → Name,
- Name → DOB

Semi Non Trivial Functional Dependencies

X → Y is called semi non-trivial when X intersect Y is not NULL.

Examples:

- AB → BC,
- AD → DC

Armstrong's Axioms in Functional Dependency

The term Armstrong axioms refer to the sound and complete set of inference rules or axioms, introduced by William W. Armstrong, that is used to test the logical implication of **functional dependencies.**

If F is a set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

1. **Axiom of reflexivity –**

   If X is a set of attributes and Y is subset of X, then X holds Y.

If X ⊇ Y then X → Y

Example:

X = {a, b, c, d, e}

Y = {a, b, c}

This property is trivial property.

2. **Axiom of augmentation –**

   The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

   If X → Y then XZ → YZ

Example:

For R(ABCD), **if** A → B then AC → BC

3. **Axiom of transitivity –**

   In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

   If X → Y and Y → Z then X → Z

Secondary Rules –

These rules can be derived from the above axioms.

1. **Union–**
   Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

   If X → Y and X → Z then X → YZ

2. **Decomposition–**
   Decomposition rule is also known as project rule. It is the reverse of union rule.This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

   If X → YZ then X → Y and X → Z

3. **Pseudo Transitivity –**
   In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

   If X → Y and YZ → W then XZ → W

Minimal Covers:

A minimal cover of a set of functional dependencies (FD) E is a minimal set of dependencies F that is equivalent to E.

The formal definition is: A set of FD F to be minimal if it satisfies the following conditions −

- Every dependency in F has a single attribute for its right-hand side.
- We cannot replace any dependency X->A in F with a dependency Y->A, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.
- We cannot remove any dependency from F and still have a set of dependencies that are equivalent to F.

Canonical cover is called minimal cover which is called the minimum set of FDs. A set of FD FC is called canonical cover of F if each FD in FC is a −

- Simple FD.
- Left reduced FD.
- Non-redundant FD.

Simple FD − X->Y is a simple FD if Y is a single attribute.

Left reduced FD − X->Y is a left reduced FD if there are no extraneous attributes in X. {extraneous attributes: Let XA->Y then, A is a extraneous attribute if X_>Y}

Non-redundant FD − X->Y is a Non-redundant FD if it cannot be derived from F- {X->y}.

Example

Consider an example to find canonical cover of F.

The given functional dependencies are as follows −

A -> BC

B -> C

A -> B

AB -> C

- Minimal cover: The minimal cover is the set of FDs which are equivalent to the given FDs.
- Canonical cover: In canonical cover, the LHS (Left Hand Side) must be unique.

First of all, we will find the minimal cover and then the canonical cover.

First step − Convert RHS attribute into singleton attribute.

A -> B

A -> C

B -> C

A -> B

AB -> C

Second step − Remove the extra LHS attribute

Find the closure of A.

A+ = {A, B, C}

So, AB -> C can be converted into A -> C

A -> B

A -> C

B -> C

A -> B

A -> C

Third step − Remove the redundant FDs.

A -> B

A -> C

Now, we will convert the above set of FDs into canonical cover.

The canonical cover for the above set of FDs will be as follows −

A -> BC

B -> C

NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of

what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.
The normal forms based on FDs:

First Normal Form (1NF):

First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. The values in an atomic domain are indivisible units.

In the *first normal form*, only single values are permitted at the intersection of each row and column; hence, there are no repeating groups.

To normalize a relation that contains a repeating group, remove the repeating group and form two new relations.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

We re-arrange the relation (table) as below, to convert it to First Normal Form.

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

Each attribute must contain only a single value from its pre-defined domain.

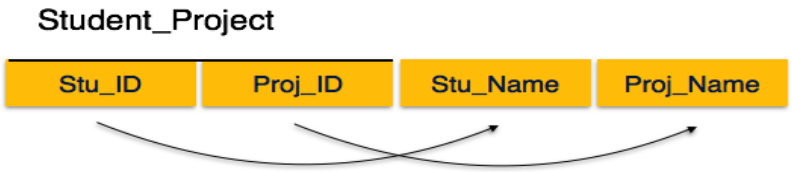
Second Normal Form (2NF):

Before we learn about the second normal form, we need to understand the following −

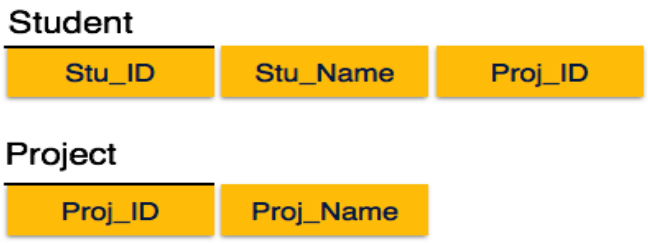
- **Prime Key attribute** − An attribute, which is a part of the candidate-key, is known as a prime attribute.

- **Non-prime attribute** − An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.

- For the *second normal form*, the relation must first be in 1NF. The relation is automatically in 2NF if, and only if, the Prime Key comprises a single attribute.
- If the relation has a composite Prime Key, then each non-key attribute must be fully dependent on the entire PK and not on a subset of the PK.
- A relation is in 2NF if it has **No Partial Dependency.**
- **Partial Dependency –** If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called **partial dependency**, which is not allowed in Second Normal Form.



We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

Third Normal Form (3NF):

To be in *third normal form*, the relation must be in second normal form. Also

- all transitive dependencies must be removed; a non-key attribute may not be functionally dependent on another non-key attribute.

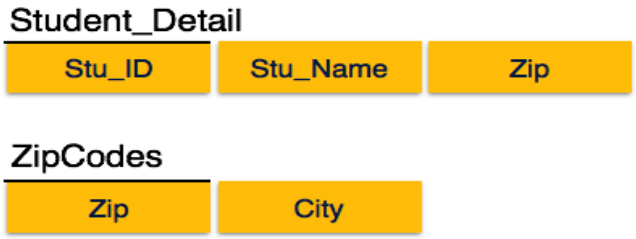
- For any non-trivial functional dependency, X → A, then either −

  - X is a superkey or,
  - A is prime attribute.

Transitive dependency – If A->B and B->C are two FDs then A->C is called transitive dependency.

We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City, so there exists **transitive dependency**.

To bring this relation into third normal form, we break the relation into two relations as follows −



Boyce-Codd Normal Form (BCNF):

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms.

A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.

In the above example, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So,

Stu_ID → Stu_Name, Zip

and

Zip → City

Which confirms that both the relations are in BCNF.

DECOMPOSITIONS

A relation in BCNF is free of redundancy and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the potential problems..

A decomposition of a relation schema *R* consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of *R* and together include all attributes in *R*.

When a relation in the relational model is not appropriate normal form then the decomposition of a relation is required. In a database, breaking down the table into multiple tables termed as decomposition.

The properties of a relational decomposition are listed below :

1. **Attribute Preservation:**
   Using functional dependencies the algorithms decompose the universal relation schema R in a set of relation schemas D = { R1, R2, ….. Rn } relational database schema, where 'D' is called the Decomposition of R.
   The attributes in R will appear in at least one relation schema Ri in the decomposition, i.e., no attribute is lost. This is called the *Attribute Preservation* condition of decomposition.
2. **Dependency Preservation:**
   If each functional dependency X->Y specified in F appears directly in one of the relation schemas Ri in the decomposition D or could be inferred from the dependencies that appear in some Ri. This is the *Dependency Preservation*.

If a relation R is decomposed into relation R1 and R2, then the dependencies of R either must be a part of R1 or R2 or must be derivable from the combination of functional dependencies of R1 and R2.

For example, suppose there is a relation R (A, B, C, D) with functional dependency set (A->BC). The relational R is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of relation R1(ABC).

3.Lossless Join:

Lossless join property is a feature of decomposition supported by normalization. It is the ability to ensure that any instance of the original relation can be identified from corresponding instances in the smaller relations.

For example:

R : relation, F : set of functional dependencies on R,

X, Y : decomposition of R,

A decomposition {R1, R2, …, Rn} of a relation R is called a lossless decomposition for R if the natural join of R1, R2, …, Rn produces exactly the relation R.

- The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.

Decomposition is lossless if:

X intersection Y -> X, that is: all attributes common to both X and Y functionally determine ALL the attributes in X.

X intersection Y -> Y, that is: all attributes common to both X and Y functionally determine ALL the attributes in Y

If X intersection Y forms a superkey of either X or Y, the decomposition of R is a lossless decomposition.

4.Lack of Data Redundancy

- Lack of Data Redundancy is also known as a **Repetition of Information.**
- The proper decomposition should not suffer from any data redundancy.
- The careless decomposition may cause a problem with the data.
- The lack of data redundancy property may be achieved by Normalization process.

UNIT-V

TRANSACTION MANAGEMENT

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes. Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

The Four Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties. Taking the initial letters of these four properties we collectively call them the **ACID Properties**. Here we try to describe them and explain them.

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B.The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs 1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred.

Consistency: If we execute a particular transaction in isolation or together with other transaction,(i.e. presumably in a multi-programming environment), the transaction will yield the same expected result.

To give better performance, every database management system supports the execution of multiple transactions at the same time, using CPU Time Sharing. Concurrently executing transactions may have to deal with the problem of sharable resources, i.e. resources that multiple transactions are trying to read/write at the same time. For example, we may have a table or a record on which tw transaction are trying to read or write at the same time. Careful mechanisms are created in order to prevent mismanagement of these sharable resources, so that there should not be any change in the way a transaction performs. A transaction which deposits Rs 100/- to account A must deposit the same amount whether it is acting alone or in conjunction with another transaction that may be trying to deposit or withdraw some amount at the same time.

Isolation: In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

Durability: It states that once a transaction has been complete the changes it has made should be permanent.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

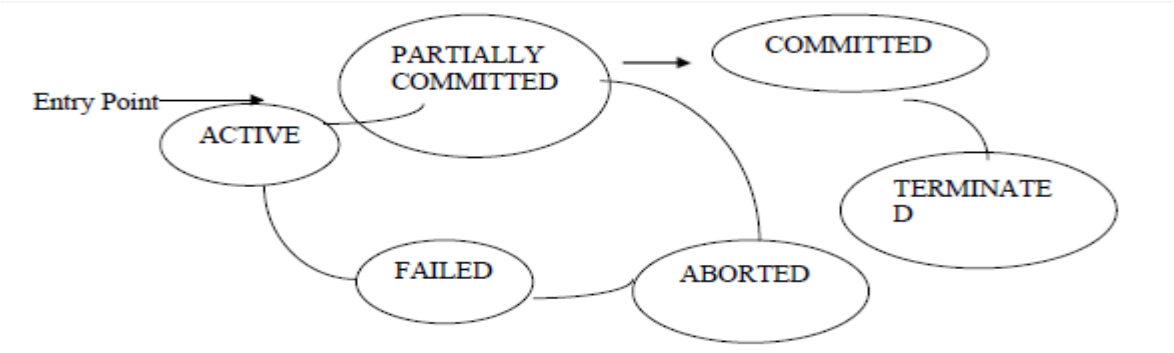
If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

Transaction States

There are the following six states in which a transaction may exist:

- **Active:** The initial state when the transaction has just started execution.
- **Partially Committed:** At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.
- **Failed:** If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.
- **Aborted:** When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.
- **Committed:** If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.
- **Terminated:** Either committed or aborted

The whole process can be described using the following diagram:



Concurrent Execution

A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- **Serial:** The transactions are executed one after another, in a non-preemptive manner.
- **Concurrent:** The transactions are executed in a preemptive, time shared method.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.

In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

T1

Read A;

A = A – 100;

Write A;

Read B;

B = B + 100;

Write B;

T2

Read A;

Temp = A * 0.1;

Read C;

C = C + Temp;

Write C;

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

T1	T2
Read A;	
A = A – 100;	
Write A;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Read B;	
B = B + 100;	
Write B;	

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.

Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.

2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.

3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.

2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.

3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item. Let us consider a schedule *S* in which there are two consecutive instructions, *I* and *J* , of transactions *Ti* and *Tj* , respectively (*i* _= *j*). If *I* and *J* refer to different data items, then we can swap *I* and *J* without affecting the results of any instruction
in the schedule. However, if *I* and *J* refer to the same data item *Q*, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

 I = read(*Q*), *J* = read(*Q*). The order of *I* and *J* does not matter, since the same value of *Q* is read by *Ti* and *Tj* , regardless of the order.

 I = read(*Q*), *J* = write(*Q*). If *I* comes before *J* , then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *J* . If *J* comes before *I*, then *Ti* reads the value of *Q* that is written by *Tj*. Thus, the order of *I* and *J* matters.

 I = write(*Q*), *J* = read(*Q*). The order of *I* and *J* matters for reasons similar to those of the previous case.

4. *I* = write(*Q*), *J* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either *Ti* or *Tj* . However, the value obtained by the next read(*Q*) instruction of *S* is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(*Q*) instruction after *I* and *J* in *S*, then the order of *I* and *J* directly affects the final value of *Q* in the database state that results from schedule *S*.

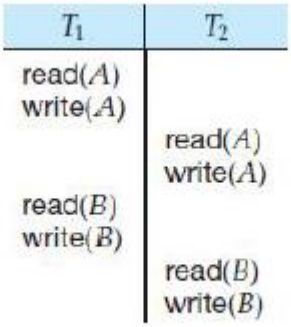


Fig: Schedule 3—showing only the read and write instructions.

We say that *I* and *J* **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3in Figure above. The write(*A*) instruction of *T*1 conflicts with the read(*A*) instruction of *T*2. However, the write(*A*) instruction of *T*2 does not conflict with the read(*B*) instruction of *T*1, because the two instructions access different data items.

Transaction Characteristics

Every transaction has three characteristics: *access mode*, *diagnostics size*, and *isolation level*. The **diagnostics size** determines the number of error conditions that can be recorded.

If the **access mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. F transactions with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure given below. In this context, *dirty read* and *unrepeatable read* are defined as usual. **Phantom** is

defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself.

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged ,and holds them until the end, according to Strict 2PL. **REPEATABLE READ** ensures that *T* reads only the changes made by committed transactions, and that no value read or written by *T* is changed by any other transaction until *T* is complete. However, *T* could experience the phantom phenomenon; for example, while *T* examines all Sailors records with *rating=1*, another transaction might add a new such Sailors record, which is missed by *T*.

A REPEATABLE READ transaction uses the same locking protocol as a SERIALIZABLE transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that *T* reads only the changes made by committed transactions, and that no value written by *T* is changed by any other transaction until *T* is complete. However, a value read by *T* may well be modified by another transaction while *T* is still in progress, and *T* is, of course, exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself - a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects, and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.

For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level, or even the READ UNCOMMITTED level, because a few incorrect or missing values will not significantly affect the result if the number of sailors is large. The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

PRECEDENCE GRAPH.

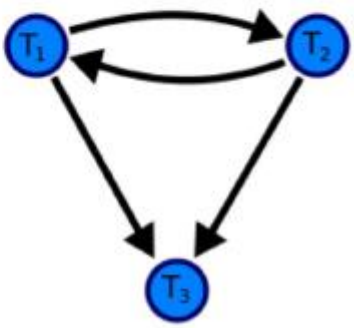
A precedence graph, also named conflict graph and serializability graph, is used in the context of concurrency control in databases.

The precedence graph for a schedule S contains:

A node for each committed transaction in S An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.

Precedence graph example

$$D = \begin{bmatrix} T1 & T2 & T3 \\ & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \\ & & \end{bmatrix}$$

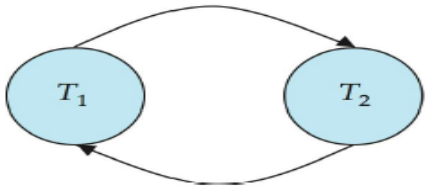


A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is not Conflict serializable.

The drawing sequence for the precedence graph:-

- For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph. So the precedence graph contains T_1, T_2, T_3
- For each case in S where T_i executes a write_item(X) then T_j executes a read_item(X), create an edge (T_i --> T_j) in the precedence graph. This occurs nowhere in the above example, as there is no read after write.
- For each case in S where T_i executes a read_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edge from T_1 to T_2.
- For each case in S where T_i executes a write_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edges from T_2 to T_1, T_1 to T_3, and T_2 to T_3.

- The schedule S is conflict serializable if the precedence graph has no cycles. As T_1 and T_2 constitute a cycle, then we cannot declare S as serializable or not and serializability has to be checked using other methods.



TESTING FOR CONFLICT SERIALIZABILITY

- 1 A schedule is conflict serializable if and only if its precedence graph is acyclic.
- 2 To test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm.Cycle-detection algorithms exist which take order n2 time, where n is the number of vertices in the graph. (Better algorithms take order n + e where e is the number of edges.)
- 3 If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. That is, a linear order consistent with the partial order of the graph.

  For example, a serializability order for the schedule (a) would be one of either (b) or (c)
- A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph.

RECOVERABLE SCHEDULES

- Recoverable schedule — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti must appear before the commit operation of Tj.
- The following schedule is not recoverable if T9 commits immediately after the read(A) operation.

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

- If T8 should abort, T9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

CASCADING ROLLBACKS

- Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable) If T10 fails, T11 and T12 must also be rolled back.
- Can lead to the undoing of a significant amount of work

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

CASCADELESS SCHEDULES

- Cascade less schedules — for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj.
- Every cascade less schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascade less.
- Example of a schedule that is NOT cascade less

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

CONCURRENCY SCHEDULE

A database must provide a mechanism that will ensure that all possible schedules are both:

- Conflict serializable.
- Recoverable and preferably cascade less
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability after it has executed is a little too late!
- Tests for serializability help us understand why a concurrency control protocol is correct
- Goal – to develop concurrency control protocols that will assure serializability.

WEEK LEVELS OF CONSISTENCY

- Some applications are willing to live with weak levels of consistency, allowingschedules that are not serializable
- E.g., a read-only transaction that wants to get an approximate total balance of all accounts
- E.g., database statistics computed for query optimization can be approximate (why?)
- Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

LEVELS OF CONSISTENCY IN SQL

- Serializable — default
- Repeatable read — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- Read committed — only committed records can be read, but successive reads of record may return different (but committed) values.
- Read uncommitted — even uncommitted records may be read.
- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

TRANSACTION DEFINITION IN SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
- Commit work commits current transaction and begins a new one.
- Rollback work causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
- Implicit commit can be turned off by a database directive
- E.g. in JDBC, connection. set Auto Commit(false);

RECOVERY SYSTEM

Failure Classification:

- Transaction failure :
- Logical errors: transaction cannot complete due to some internal error condition
- System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

- System crash: a power failure or other hardware or software failure causes the system to crash.
- Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted as result of a system crash
- Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure: a head crash or similar disk failure destroys all or part of disk storage
- Destruction is assumed to be detectable: disk drives use checksums to detect failures

RECOVERY ALGORITHMS

- Consider transaction Ti that transfers $50 from account A to account B
- Two updates: subtract 50 from A and add 50 to B Transaction Ti requires updates to A and B to be output to the database.
- A failure may occur after one of these modifications have been made but before both of them are made.
- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
- Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
-

1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures

2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

STORAGE STRUCTURE

- Volatile storage:
- Does not survive system crashes
- Examples: main memory, cache memory
- Nonvolatile storage:
- Survives system crashes
- Examples: disk, tape, flash memory,
- Non-Volatile (battery backed up) RAM

- But may still fail, losing data
- Stable storage:
- a mythical form of storage that survives all failures
- Approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
- copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies.
- Block transfer can result in
- Successful completion
- Partial failure: destination block has incorrect information
- Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
- Execute output operation as follows (assuming two copies of each block):

1. Write the information onto the first physical block.
2. When the first write successfully completes, write the same information onto the second physical block.
3. The output is completed only after the second write successfully completes.

- Copies of a block may differ due to failure during output operation. To recover from failure: First find inconsistent blocks:

1. Expensive solution: Compare the two copies of every disk block.
2. Better solution:

- Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
- Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
- Used in hardware RAID systems

If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

DATA ACCESS

- Physical blocks are those blocks residing on the disk.
- System buffer blocks are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
- input(B) transfers the physical block B to main memory.
- output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Each transaction Ti has its private work-area in which local copies of all data items accessed and updated by it are kept.
- Ti's local copy of a data item X is denoted by xi.
- BX denotes block containing X
- Transferring data items between system buffer blocks and its private work-area done by:
- read(X) assigns the value of data item X to the local variable xi.
- write(X) assigns the value of local variable xi to data item {X} in the buffer block.
- Transactions
- Must perform read(X) before accessing X for the first time (subsequent reads can be from local copy)
- The write(X) can be executed at any time before the transaction commits
- Note that output(BX) need not immediately follow write(X). System can perform the output operation when it seems fit.

Lock-Based Protocols

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

Data items can be locked in two modes :

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

2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction. Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

1) A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

2) Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

3) If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

T2:

lock-S*(A)*;

read *(A)*;

unlock*(A)*;

lock-S*(B)*;

read *(B)*;

unlock*(B)*;

display*(A+B)*

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Consider the partial schedule

T_3	T_4
lock-x (*B*)	
read (*B*)	
B := *B* - 50	
write (*B*)	
	lock-s (*A*)
	read (*A*)
	lock-s (*B*)
lock-x (*A*)	

Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A. Such a situation is called a **deadlock**.

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

2. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

3.Starvation is also possible if concurrency control manager is badly designed. For example:

- A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
- The same transaction is repeatedly rolled back due to deadlocks.

4.Concurrency control manager can be designed to prevent starvation.

THE TWO-PHASE LOCKING PROTOCOL

1.This is a protocol which ensures conflict-serializable schedules.

2.Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

3.Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

4.The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

5.Two-phase locking *does not* ensure freedom from deadlocks

6.Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.

7.Rigorous two-phase locking is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

8.There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

9.However, in the absence of extra information (e.g., ordering of access to data), twophase

locking is needed for conflict serializability in the following sense: Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

TIMESTAMP-BASED PROTOCOLS

1.Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).

2.The protocol manages concurrent execution such that the time-stamps determine the serializability order.

3.In order to assure such behavior, the protocol maintains for each data Q two timestamp values:

- W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.
- R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.

4.The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

5.Suppose a transaction T_i issues a **read**(Q)

- If TS(T_i) □ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. n Hence, the **read** operation is rejected, and T_i is rolled back.
- If TS(T_i)□ **W**-timestamp(Q), then the **read** operation is executed, and Rtimestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

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

- If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. n Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. n Hence, this **write** operation is rejected, and T_i is rolled back.
- Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

1.We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol i.e., Timestamp ordering Protocol . Let us

consider schedule 4 of Figure below, and apply the timestamp-ordering protocol. Since *T*27 starts before *T*28, we shall assume that TS(*T*27) < TS(*T*28). The read(*Q*) operation of *T*2 succeeds, as does the write(*Q*) operation of *T*28. When *T*27 attempts its write(*Q*) operation, we find that TS(*T*27) < W-timestamp(*Q*), since Wtimestamp(*Q*) = TS(*T*28). Thus, the write(*Q*) by *T*27 is rejected and transaction *T*27 must be rolled back.

2.Although the rollback of *T*27 is required by the timestamp-ordering protocol, it is unnecessary. Since *T*28 has already written *Q*, the value that *T*27 is attempting to write is one that will never need to be read. Any transaction *Ti* with TS(*Ti*) < TS(*T*28) that attempts a read(*Q*)will be rolled back, since TS(*Ti*)<W-timestamp(*Q*).

3.Any transaction *Tj* with TS(*Tj*) > TS(*T*28) must read the value of *Q* written by *T*28, rather than the value that *T*27 is attempting to write. This observation leads to a modified version, of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the time stamp order in protocol.

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this:
Suppose that transaction *Ti* issues write(*Q*).

1. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.

2. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and setsW-timestamp(*Q*) to TS(*Ti*).

VALIDATION-BASED PROTOCOLS

1) Phases in Validation-Based Protocols of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.

2) Validation phase. The validation test is applied to transaction Ti. This determines whether Ti is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.

3) Write phase. If the validation test succeeds for transaction Ti, the temporary local variables that hold the results of any write operations performed by Ti are copied to the database. Read-only transactions omit this phase.

MODES IN VALIDATION-BASED PROTOCOLS

1. Start(Ti)

2. Validation(Ti)

3. Finish

MULTIPLE GRANULARITY.

multiple granularity locking (MGL) is a locking method used in database management systems (DBMS) and relational databases. In MGL, locks are set on objects that contain other objects. MGL exploits the hierarchical nature of the contains relationship. For example, a database may have files, which contain pages, which further contain records. This can be thought of as a tree of objects, where each node contains its children. A lock on such as a shared or exclusive lock locks the targeted node as well

as all of its descendants. Multiple granularity locking is usually used with non-strict two-phase locking to guarantee serializability. The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

- Transaction *Ti* must observe the lock-compatibility function of Figure above.
- Transaction *Ti* must lock the root of the tree first, and can lock it in any mode.
- Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.
- Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
- Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).
- Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.