Chapter 1. Introduction to Database Concepts

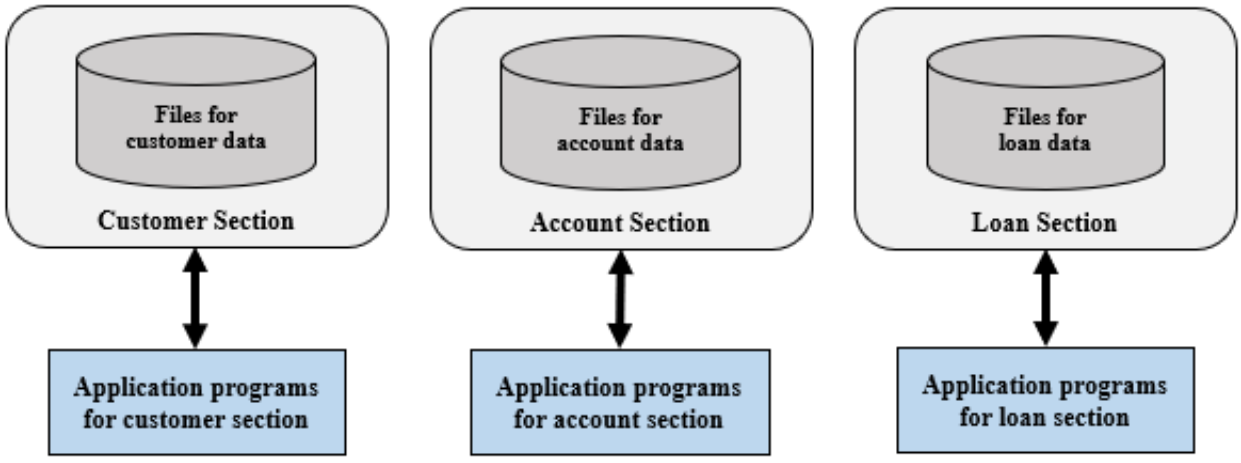
File Processing System

In earlier days, data was stored manually, using pen and paper. But after discovery of computer, the same task is done by using files.

A computer **file** is a resource which uniquely records data, in a storage device in a computer. There are various formats in which data can be stored. e.g. text files as.txt while pictures as .png files.

In Computer Science, file processing system (FPS) is a way of storing, retrieving and manipulating data stored permanently in various file in a computer. FPS may use files such as .txt, .jpg, .docx, or even structured datatypes such as .html or .xml. To manipulate data stored in these files, a number of application programs needs to be written at the request of the users in the organization like arranging monthly sales data or printing monthly reports of sales. New applications are added to the system as the need arises.

Example- File processing system for traditional bank



Major disadvantages of file-processing system:

In the early days of their use, file processing systems were, **major advances** in data. management. Even though, FPS are cost friendly and easy to understand and use, their major **disadvantages are -**

1. Data redundancy and inconsistency
2. Difficulty in accessing data

3. Data isolation

4. Integrity problems

5. Atomicity of updates

6. Concurrent access by multiple users

1. Data redundancy and inconsistency:

Data is stored in **multiple file formats** (.csv, .txt or .doc)

Application programs to access the data in different files may be written in different languages (C or C++).

Duplication of information in different files in different formats. (DOB as dd-mm-yy or mm-dd-yyyy) Increases **storage cost.**

Data inconsistency (change in address of customer may not get reflected in all sections).

2. Difficulty in accessing data

Need to write a **new program to carry out each new task**.

3. Data isolation

Data isolation is a property that determines when and how changes made by one operation become visible to other **concurrent users and systems**.

This is a problem because writing application programs is difficult as data is scattered in multiple files (account and loan in different files) in multiple formats.

4. Integrity problems

Data integrity refers to the maintenance and assurance that the data in a database are **correct and consistent**.

Factors to consider when addressing this issue are:

- Data values must satisfy **certain consistency constraints** that are specified in the application programs. For example - account balance > 0
- It is difficult to make changes to the application programs in order **to enforce new constraints**. For example - eligibility for home loans in bank

5. Atomicity of updates

Failures in maintaining **atomicity** in transactions, may leave database in an **inconsistent (incorrect) state with partial updates** carried out.

Example - Transfer of funds from account A to account B should either complete or not happen at all.

6. Concurrent access by multiple users

- Concurrency is the ability to allow **multiple users access to the same record simultaneously** without adversely affecting transaction processing.
- Typically, in a file-based system, when an application opens a file, that **file is locked**. This means that no one else has access to the file at the same time.

Data, Database and Database Management System

Data

Data can be **facts** related to any object in consideration. It is a set of different **symbols and characters**.

Example: name, age, height, weight, etc. are some data related to person. A picture, image, file, pdf, etc. can also be considered data.

Database

A database is a **organized** collection of **inter-related** data which represents **some aspect of the real world**.

Example: Student database, Employee database, Patient database, Book database

Database Management System

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

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

Examples of popular DBMS:

- MySql
- Oracle
- SQL Server
- IBM DB2
- PostgreSQL
- Amazon SimpleDB (cloud based) etc.

Characteristics of databases

Traditionally, data was organized in file formats. DBMS was a new concept then, and all the research was done to make it overcome the deficiencies in traditional style of data management.

Following are the Characteristics of modern DBMS-

Real-world entity

A modern DBMS is more realistic and uses real-world entities to design its architecture. It uses the behaviour and attributes too. For example, a college database may use students as an entity and their age as an attribute.

Relation-based tables

DBMS allows entities and relations among them to form tables. A user can understand the architecture of a database just by looking at the table names.

Isolation of data and application

A database system is entirely different than its data. A database is an active entity, whereas data is said to be passive, on which the database works and organizes. DBMS also stores metadata, which is data about data, to ease its own process.

Less redundancy

DBMS follows the rules of normalization, which splits a relation when any of its attributes is having redundancy in values. Normalization is a mathematically rich and scientific process that reduces data redundancy.

Consistency

Consistency is a state where every relation in a database remains consistent. There exist methods and techniques, which can detect attempt of leaving database in inconsistent state. A DBMS can provide greater consistency as compared to earlier forms of data storing applications like file-processing systems.

Query Language

DBMS is equipped with query language, which makes it more efficient to retrieve and manipulate data. A user can apply as many and as different filtering options as required to retrieve a set of data. Traditionally it was not possible where file-processing system was used.

ACID Properties

DBMS follows the concepts of **A**tomicity, **C**onsistency, **I**solation, and **D**urability (normally shortened as ACID). These concepts are applied on transactions, which manipulate data in a database. ACID properties help the database stay healthy in multi-transactional environments and in case of failure.

Multiuser and Concurrent Access

DBMS supports multi-user environment and allows them to access and manipulate data in parallel. Though there are restrictions on transactions when users attempt to handle the same data item, but users are always unaware of them.

Multiple views

DBMS offers multiple views for different users. A user who is in the Sales department will have a different view of database than a person working in the Production department. This feature enables the users to have a concentrate view of the database according to their requirements.

Security

Features like multiple views offer security to some extent where users are unable to access data of other users and departments. DBMS offers methods to impose constraints while entering data into the database and retrieving the same at a later stage. DBMS offers many different levels of security features, which enables multiple users to have different views with different features. For example, a user in the Sales department cannot see the data that belongs

to the Purchase department. Additionally, it can also be managed how much data of the Sales department should be displayed to the user. Since a DBMS is not saved on the disk as traditional file systems, it is very hard for miscreants to break the code.

Database System Applications

Databases are widely used. Here are some representative applications:

- **Enterprise Information**
  - Sales: For customer, product, and purchase information.
  - Accounting: For payments, receipts, account balances, assets and other accounting information.
- **Human resources:**

  For information about employees, salaries, payroll taxes, and benefits, and for generation of 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 and Finance**

Banking:

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

Credit card transactions:

For purchases on credit cards and generation of monthly statements.

Finance:

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

• **Universities:**

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

• **Airlines:** For reservations and schedule information.

Airlines were among the first to use databases in a geographically distributed manner.

• **Telecommunication:**

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

As the list illustrates, databases form an essential part of every enterprise today, storing not only types of information that are common to most enterprises, but also information that is specific to the category of the enterprise.

The importance of database systems can be judged in another way, today, database system vendors like Oracle are among the largest software companies in the world, and database systems form an important part of the product line of Microsoft and IBM.

File system vs **Database Management System (DBMS)**

File System	Database Management System (DBMS)
1. It is a software system that manages and controls the data files in a computer system.	1. It is a software system used for creating and managing the databases. DBMS provides a systematic way to access, update, and delete data.
2. File system does not support multi-user access.	2. Database Management System supports multi-user access.
3. Data consistency is less in the file system.	3. Data consistency is more due to the use of normalization.
4. File system is not secured.	4. Database Management System is highly secured.
5. File system is used for storing the unstructured data.	5. Database management system is used for storing the structured data.
6. In the file system, data redundancy is high.	6. In DBMS, Data redundancy is low.
7. No data backup and recovery process is present in a file system.	7. There is a backup recovery for data in DBMS.
8. Handling of a file system is easy.	8. Handling a DBMS is complex.
9. Cost of a file system is less than the DBMS.	9. Cost of database management system is more than the file system.

10. If one application fails, it does not affect other application in a system.	10. If the database fails, it affects all application which depends on it.
11. In file system, data cannot be shared because it is distributed in different files.	11. In DBMS, data can be shared as it is stored at one place in a database.
12. These system does not provide concurrency facility.	12. This system provides concurrency facility.
13. **Example:** NTFS (New technology file system)	13. **Example:** Oracle, MySQL, MS SQL Server, DB2, Microsoft Access, etc.

(https://www.tutorialandexample.com/difference-between-file-system-and-dbms/)

View of Data

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

Data Abstraction

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

• **Physical level**

The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

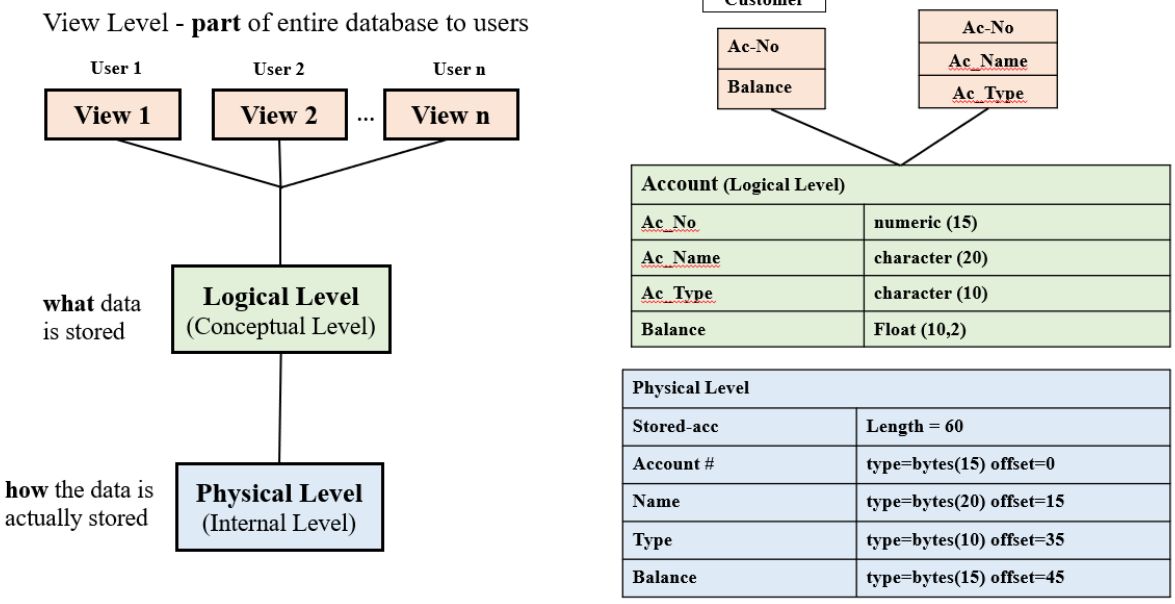
• **Logical level**

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**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level**

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.

Figure below shows the relationship among the three levels of abstraction.



Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program.

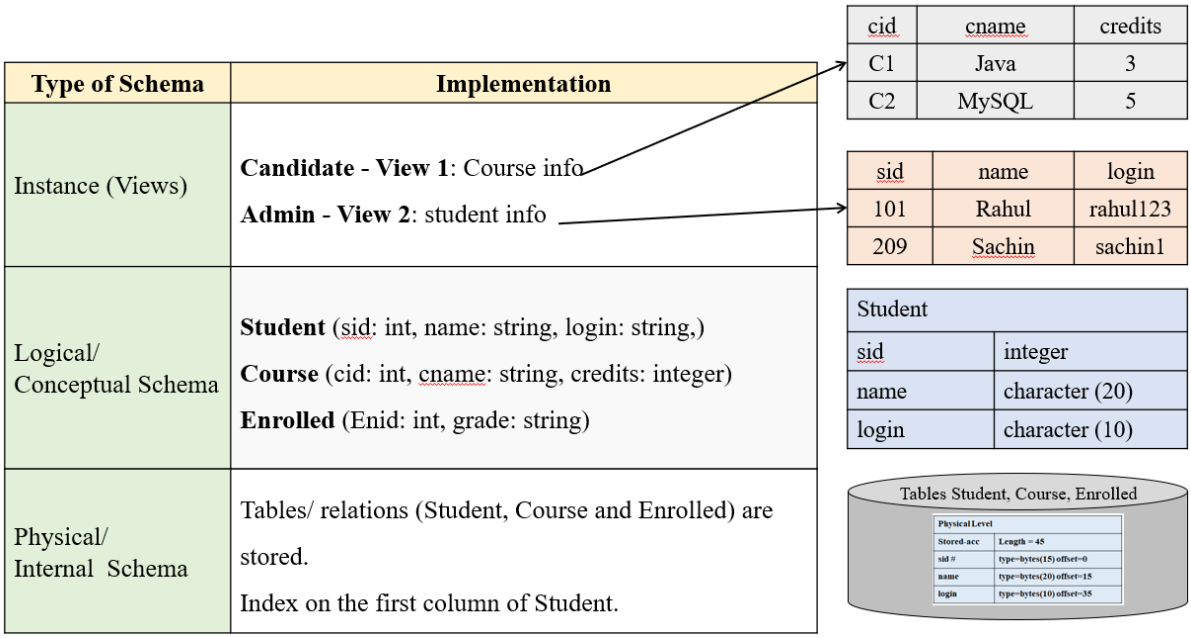
Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an instance of a database schema.

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

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level.

A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.



Data Models

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

There are a number of different data models that we shall cover in the text.

Categories of data models are

• **Relational Model**

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model.

Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

• **Entity-Relationship Model**

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

• **Object-Based Data Model**

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

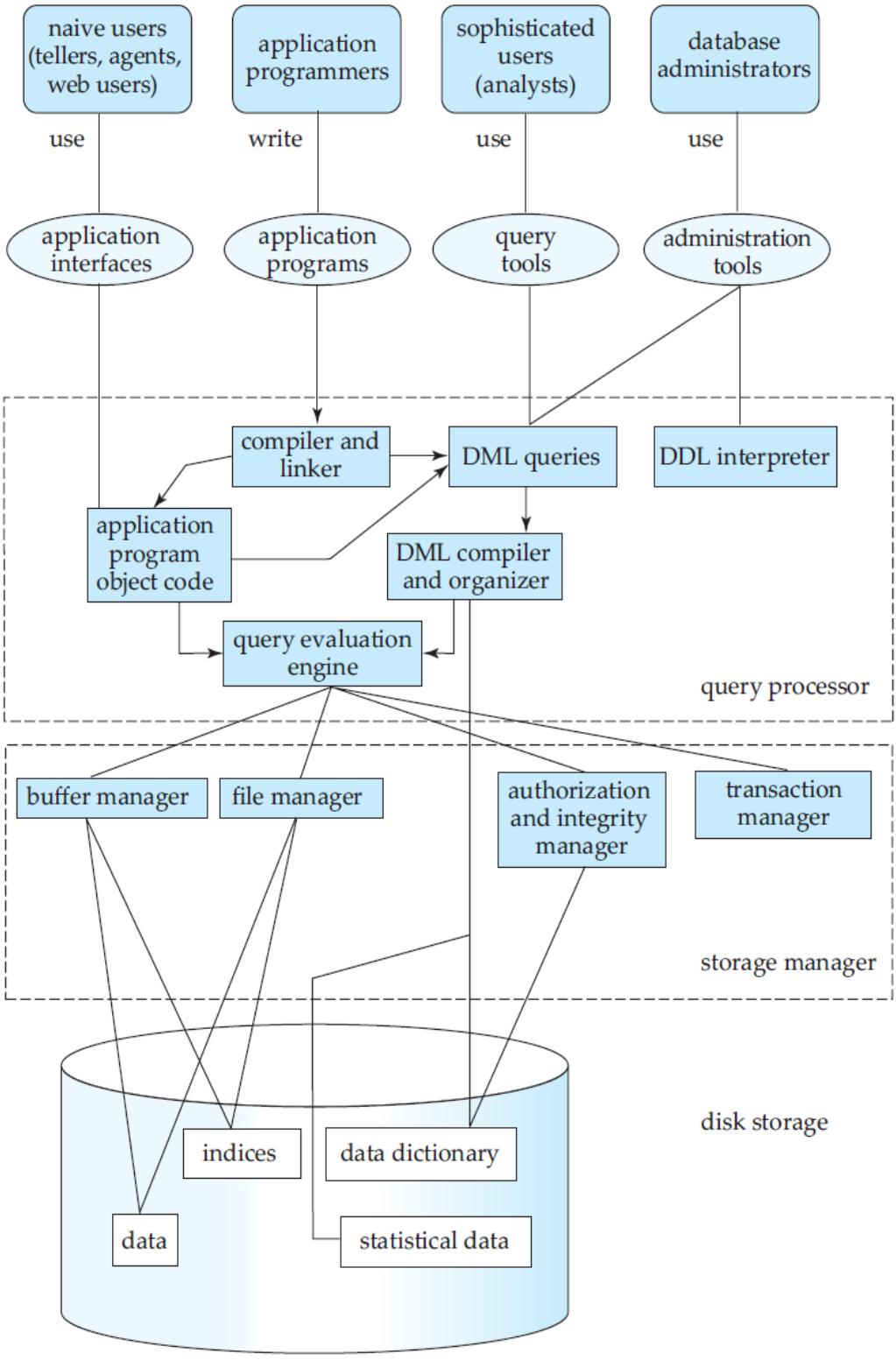
• **Semi-structured Data Model**

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

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

Database Architecture

Database Architecture represents the various components of a database system and the connections among them.



A database system has several **subsystems** like

- storage manager subsystem
- query processor subsystem
- transaction management and
- disk storage

Query processor subsystem

Query processor **compiles and executes DDL and DML statements**.

DDL interpreter – It interprets DDL statements and records the definitions in the data dictionary.

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

The DML compiler performs query optimization; that is, it picks the lowest cost evaluation plan from among the various alternatives.

Query evaluation engine – It executes low-level instructions generated by the DML compiler.

Storage manager

It provides the **interface** between the stored in the database and the application programs and queries submitted to the system.

The **storage manager** is responsible to the following tasks:

- Interaction with the OS file manager
- Efficient storing, retrieving and updating of data

The storage manager components are:

- Authorization and integrity manager
- Transaction manager
- File manager
- Buffer manager

Transaction management

Transaction manager performs two activities -

- Transaction-management ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- Concurrency-control management controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Disk storage

Disk storage has following data structures are part of the physical system implementation and implemented by the **storage manager-**

- **Data files** - store the **database itself**
- **Data dictionary** - stores **metadata** about the structure of the database, in particular the schema of the database.
- **Indices** - can provide **fast access** to data items. A database index provides pointers to those data items that hold a particular value.
- **Statistical data** - maintains **statistics** about different query execution plans and time required for execution

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.

Database applications are typically broken-up into a **front-end part** that runs at client machines and a part that runs at the **back end**. In two-tier architectures, the front end directly communicates with a database running at the back end. In three-tier architectures, the back-end part is itself broken up into an application server and a database server.

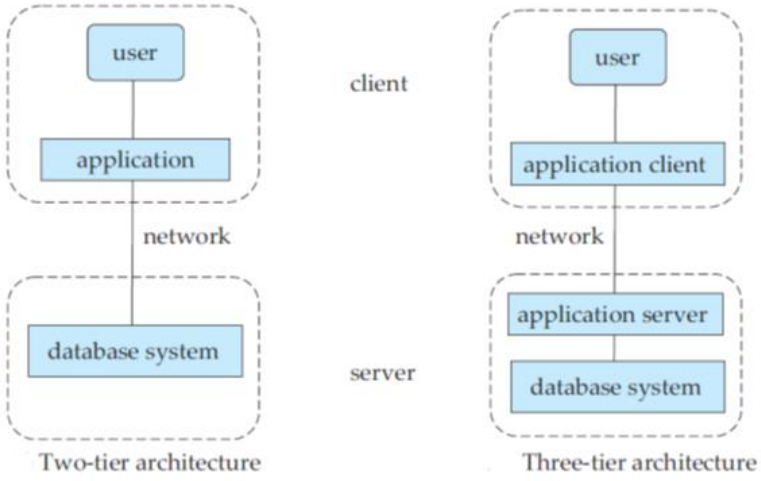
Database applications architectures

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between **client** machines, on which remote database users work, and **server** machines, on which the database system runs.

Database applications are usually partitioned into **two or three parts.**

In a **two-tier architecture**, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in a **three-tier architecture**, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an **application server**, usually through a forms interface. The application server in turn communicates with a database system to access data. The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the WorldWideWeb (WWW).



Database Users and Administrators

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

Database User types

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 clerk in the university who needs to add a new instructor to department A invokes a program called new hire. This program asks the clerk for the name of the new instructor, her new ID, the name of the department (that is, A), and the salary.

The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read reports generated from the database.

• **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. Rapid application development (RAD) tools are tools that enable an application programmer to construct forms and reports with minimal programming effort.

• **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.

• **Database Administrator (DBA)**

DBA is a type of user who has central control over the database system.

Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator (DBA)**.

Functions/duties/responsibilities of a DBA

- **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition**.
- **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:

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

Reference: Database System Concepts, Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Sixth Edition Mcgraw-Hill Publication

Chapter 2
Entity–Relationship Data Model

OVERVIEW OF THE DESIGN PROCESS

The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data. The needs of the users play a central role in the design process.

DESIGN PHASES

For small applications, it may be feasible for a database designer who understands the application requirements to decide directly on the relations to be created, their attributes, and constraints on the relations. However, such a direct design process is **difficult for real-world applications**, since they are often highly complex.

Often no one person understands the complete data needs of an application.

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

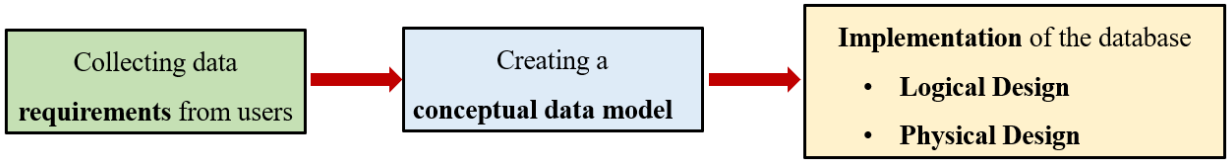
• The initial phase of database design is to characterize fully the **data needs of the prospective database users**. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements. While there are techniques for diagrammatically representing user requirements, in this chapter we restrict ourselves to textual descriptions of user requirements.

• Next, the designer **chooses a data model** and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise. The entity-relationship model, which we study in the rest of this chapter, is typically used to represent the conceptual design. Stated in terms of the entity-relationship model, the conceptual schema specifies the entities that are represented in the database, the attributes of the entities, the relationships among the entities, and constraints on the entities and relationships. Typically, the conceptual-design phase results in the creation of an entity-relationship diagram that provides a graphic representation of the schema.

The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. She can also examine the design to remove any redundant features. Her focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

• A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or

updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.



FINAL DESIGN PHASES

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

- In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The implementation data model is typically the relational data model, and this step typically consists of mapping the conceptual schema defined using the entity-relationship model into a relation schema.
- Finally, the designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.
  The physical schema of a database can be changed relatively easily after an application has been built. However, changes to the logical schema are usually harder to carry out, since they may affect a number of queries and updates scattered across application code. It is therefore important to carry out the database design phase with care, before building the rest of the database application.

DESIGN ALTERNATIVES

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

In a university database, examples of entities would include *faculty*, students, departments, courses, and course offerings. The various entities are related to each other in a variety of ways, all of which need to be captured in the database design. For example, a student takes a course offering, while an *faculty* teaches a course offering; teaches and takes are examples of relationships between entities.

TWO MAJOR PITFALLS IN DESIGN:

1. REDUNDANCY:

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

Redundancy can also occur in a relational schema. In the university example we have used so far, we have a relation with section information and a separate relation with course information. Suppose that instead we have a single relation where we repeat all of the course information (course id, title, dept name, credits) once for each section (offering) of the course. Clearly, information about courses would then be stored redundantly.

The biggest problem with such redundant representation of information is that the copies of a piece of information can become inconsistent if the information is updated without taking precautions to update all copies of the information. For example, different offerings of a course may have the same course identifier, but may have different titles. It would then become unclear what the correct title of the course is. Ideally, information should appear in exactly one place.

2. INCOMPLETENESS:

A bad design may make certain aspects of the enterprise difficult or impossible to model. For example, suppose that, as in case (1) above, we only had entities corresponding to course offering, without having an entity corresponding to courses.

ENTITY-RELATIONSHIP MODEL

The **entity-relationship** (**E-R**) data model was developed to facilitate database design by allowing specification of an enterprise schema that **represents the overall logical structure of a database**. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model.

The E-R data model employs **three basic concepts**: entity sets, relationship sets, and attributes. The E-R model also has an associated diagrammatic representation, the E-R diagram.

1. ENTITY SETS

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a *person id* property whose value uniquely identifies that person. Thus, the value 677-89-9011 for *person id* would uniquely identify one particular person in the university. Similarly, courses can be thought of as entities, and *course id* uniquely identifies a course entity in the university. An entity may be concrete, such as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are *faculty* at a given university, for example, can be defined as the entity set *faculty*. Similarly, the entity set *student* might represent the set of all students in the university.

2. ATTRIBUTES

An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *faculty* entity set are *ID*, *name*, *dept name*, and *salary*.

In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we omit them to keep our examples simple. Possible attributes of the *course* entity set are *course id*, *title*, *dept name*, and *credits*.

Each entity has a **value** for each of its attributes. For instance, a particular *faculty* entity may have the value 12121 for *ID*, the value Mahesh for *name*, the value Finance for *dept name*, and the value 90000 for *salary*.

The *ID* attribute is used to identify *faculty* uniquely, since there may be more than one *faculty* with the same name.
In the United States, many enterprises find it convenient to use the *social-security* number of a person as an attribute whose value uniquely identifies the person while in India, we may use the *Aadhar or PAN number* of a person as an attribute whose value uniquely identifies the person.
In general, the enterprise would have to create and assign a unique identifier for each *faculty*.
A database thus includes a collection of entity sets, each of which contains any number of entities of the same type.

Student
Student-ID
Student_Name
Student_Class

Student_ID	Student_Name	Student_Class
S-101	Sachin Tendulkar	First Year
S-102	Rahul Dravid	Final Year
S-109	Virat Kohli	Final Year

RELATIONSHIP SETS

A **relationship** is an association among several entities.
For example, we can define a relationship *Teaches* that associates faculty XYZ with student Sachin. This relationship specifies that Prof. XYZ is a teacher of student Sachin.
A **relationship set** is a set of relationships of the same type. Formally, it is a mathematical relation on $n \geq 2$ (possibly non-distinct) entity sets.
If $E1, E2, \ldots, En$ are entity sets,
then a relationship set R is a subset of $\{(e1, e2, \ldots, en) \mid e1 \in E1, e2 \in E2, \ldots, en \in En\}$
where $(e1, e2, \ldots, en)$ is a relationship.

Faculty

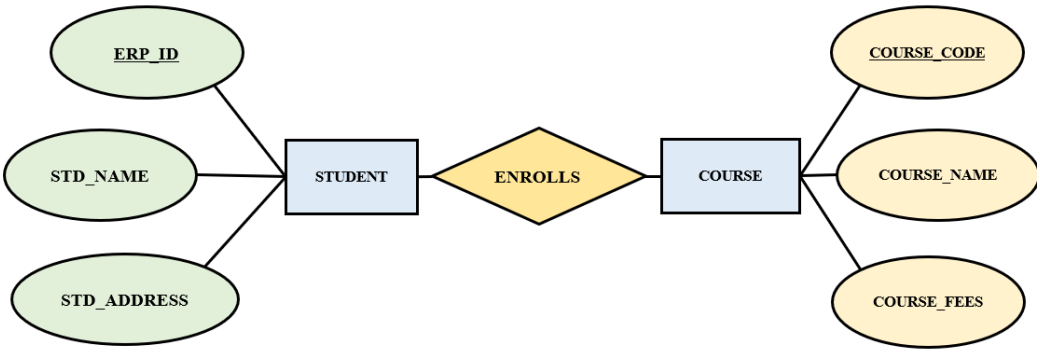
EMP-ID	FACULTY_NAME	DEPT
T901	Prof. ABC	Computer
T902	Prof. XYZ	Mechanical
T903	Prof. PQR	Civil

Student

ERP-ID	STUDENT_NAME	CLASS
S101	Sachin	SE
S102	Rahul	TE
S103	Ashwin	TE

As another example, consider the two entity sets *student* and *course*. We can define the relationship set *takes* to denote the association between a student and the course in which that student is enrolled.
The association between entity sets is referred to as **participation**; that is, the entity sets $E1, E2, \ldots, En$ **participate** in relationship set R.

Example: **STUDENT and COURSE** entities **participate** in a relationship instance of **ENROLLS**.



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

Example: relationship instance of **ENROLLS** for entities STUDENT and COURSE can be shown as

STUDENT

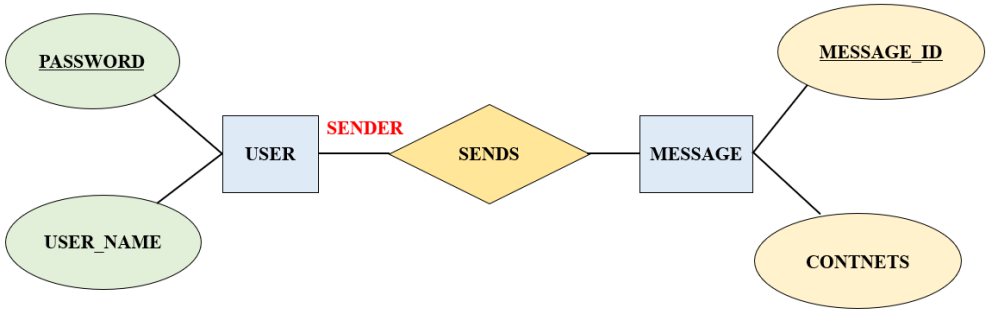
ERP-ID	STD_NAME	STD_ADDRESS
201901	Sachin	Bandra
201902	Rahul	Vashi
201903	Ashwin	Thane

ENROLLS

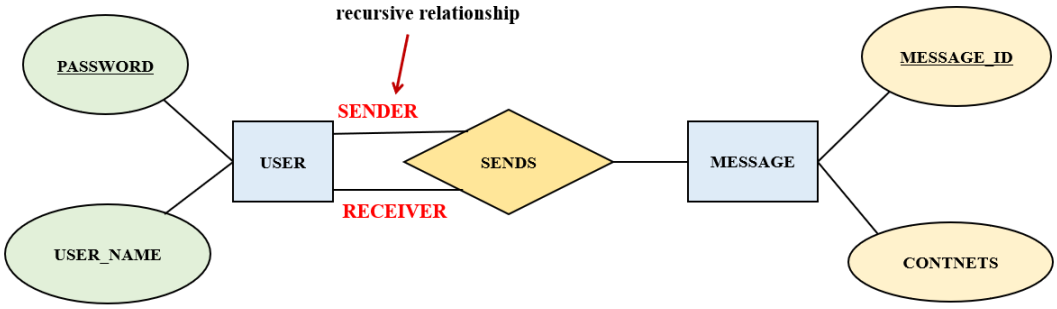
COURSE

COURSE_CODE	COURSE_NAME	COURSE_FEES
C1	JAVA	3000
C2	MySQL	3500
C3	Networking	2500

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



In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance.

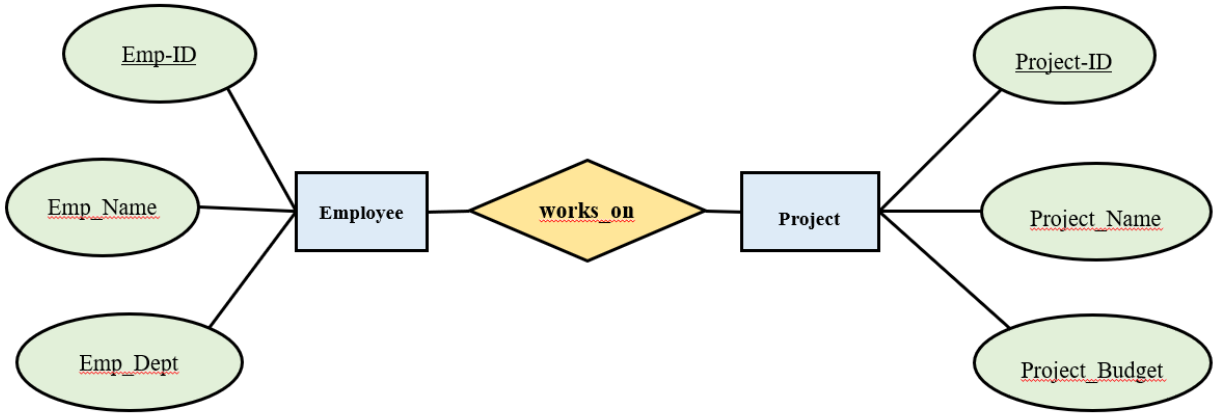


DEGREE OF RELATIONSHIP SET

The number of entity sets that participate in a relationship set is the **degree** of the relationship set.

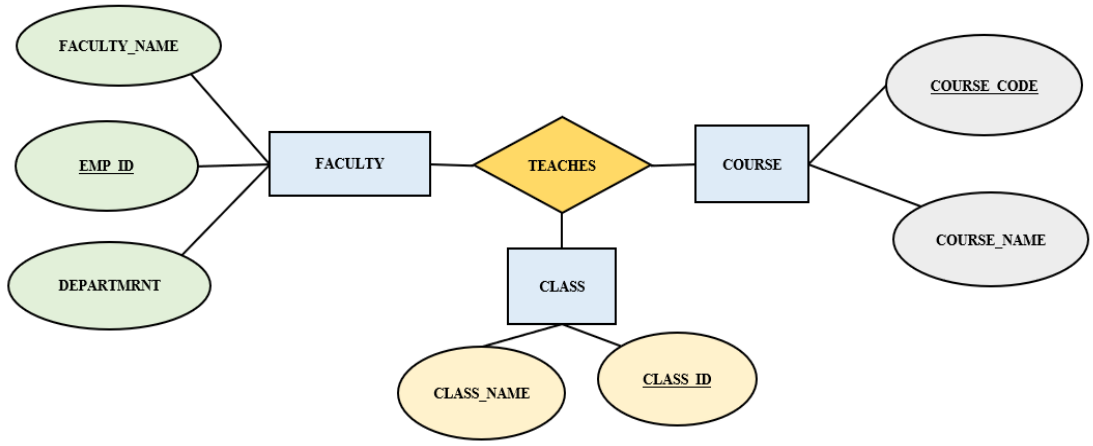
1. **Degree 2** or **Binary** relationship set
2. **Degree 3** or **Ternary** relationship set
3. **Degree n** or **n-ary** relationship set

Example: **works_on** is a **binary relationship** where **Employee and Project are participating entities**.



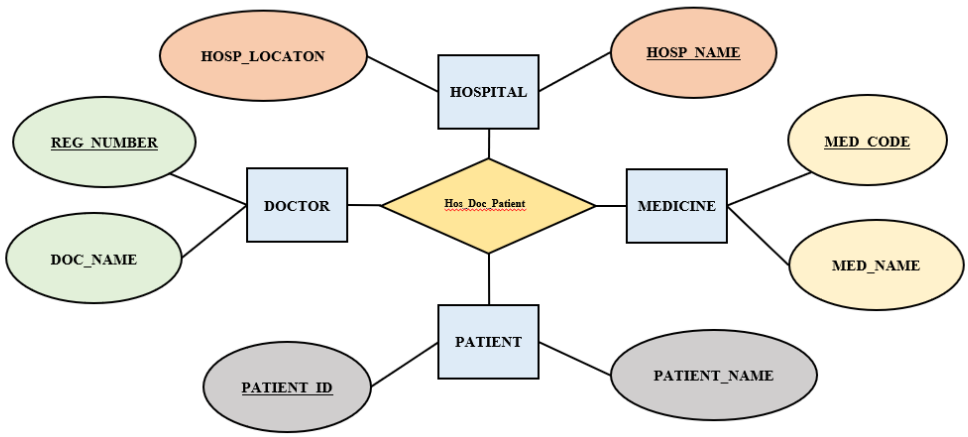
Example : Consider three entity sets 'Faculty', 'Course', and 'Class'.

The relationship between these entities is defined as the **Faculty teaching a particular Course to a particular Class**.



Example: Consider three entity sets 'Doctor', 'Patient', and 'Medicine' and 'Hospital'.

The relationship between these entities is defined as **patient visits a hospital**, the **doctor in hospital treats the patient, also prescribes a medicine**.



ATTRIBUTES

For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute. The domain of attribute *course id* might be the set of all text strings of a certain length. Similarly, the domain of attribute *semester* might be strings from the set *{*Fall, Winter, Spring, Summer*}*.

Formally, an attribute of an entity set is a function that maps from the entity set into a domain. Since an entity set may have several attributes, each entity can be described by a set of (attribute, data value) pairs, one pair for each attribute of the entity set. For example, a particular *instructor* entity may be described by the set *{*(*ID*, 76766), (*name*, Crick), (*dept name*, Biology), (*salary*, 72000)*}*, meaning that the entity describes a person named Crick whose instructor *ID* is 76766, who is a member of the Biology department with salary of $72,000. We can see, at this point, an integration of the abstract schema with the actual enterprise being modeled. The attribute values describing an entity constitute a significant portion of the data stored in the database.

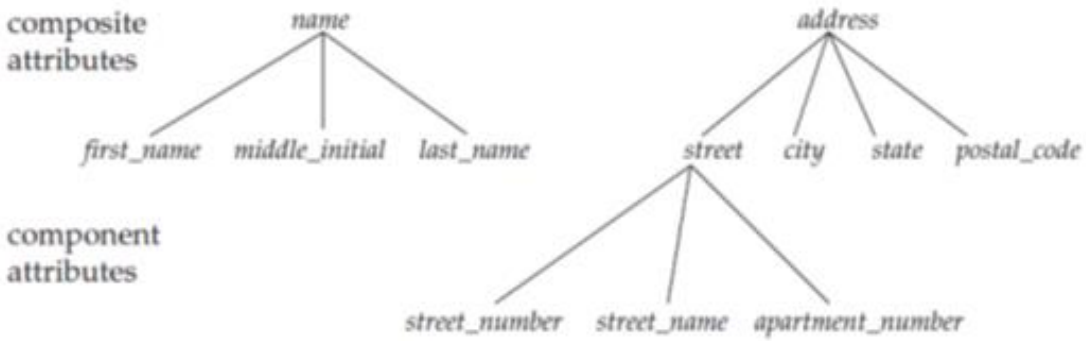
An attribute, as used in the E-R model, can be characterized by the following attribute types.

SIMPLE AND COMPOSITE ATTRIBUTES-

In our examples thus far, the attributes have been simple; that is, they have not been divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute *name* could be structured as a composite

attribute consisting of *first name*, *middle initial*, and *last name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Suppose we were to add an address to the *student* entity-set. The address can be defined as the composite attribute *address* with the attribute's *street*, *city*, *state*, and *zip code*.3 Composite attributes help us to group together related attributes, making the modeling cleaner.

Note also that a composite attribute may appear as a hierarchy. In the composite attribute *address*, its component attribute *street* can be further divided into *street number*, *street name*, and *apartment number*. Figure 7.4 depicts these examples of composite attributes for the *instructor* entity set.



SINGLE-VALUED AND MULTIVALUED ATTRIBUTES-

The attributes in our examples all have a single value for a particular entity. For instance, the *student ID* attribute for a specific student entity refers to only one student *ID*. Such attributes are said to be **single valued**.

There may be instances where an attribute has a set of values for a specific entity. Suppose we add to the *instructor* entity set a *phone number* attribute. An *instructor* may have zero, one, or several phone numbers, and different instructors may have different numbers of phones.This type of attribute is said to be **multivalued**.

As another example, we could add to the *instructor* entity set an attribute *dependent name* listing all the dependents. This attribute would be multivalued, since any particular instructor may have zero, one, or more dependents.

To denote that an attribute is multivalued, we enclose it in braces, for example {*phone number*} or {*dependent name*}.

Where appropriate, upper and lower bounds may be placed on the number of values in a multivalued attribute. For example, a university may limit the number of phone numbers recorded for a single instructor to two. Placing bounds in this case expresses that the *phone number* attribute of the *instructor* entity set may have between zero and two values.

DERIVED ATTRIBUTE-

The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the *instructor* entity set has an attribute *students advised*, which represents how many students an instructor advises. We can derive the value for this attribute by counting the number of *student* entities associated with that instructor.

As another example, suppose that the *instructor* entity set has an attribute *age* that indicates the instructor's age. If the *instructor* entity set also has an attribute *date of birth*, we can calculate *age* from *date of birth* and the current date. Thus, *age* is a derived attribute. In this case, *date of birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored but is computed when required.

NULL VALUE-

An attribute takes a **null** value when an entity does not have a value for it. The *null* value may indicate "not applicable"—that is, that the value does not exist for the entity. For example, one may have no middle name. *Null* can also designate that an attribute value is unknown. An unknown value may be either *missing* (the value does exist, but we do not have that information) or *not known* (we do not know whether or not the value actually exists).

For instance, if the *name* value for a particular instructor is *null*, we assume that the value is missing, since every instructor must have a name. A null value for the *apartment number* attribute could mean that the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what it is (missing), or that we do not know whether or not an apartment number is part of the instructor's address (unknown).

CONSTRAINTS

An E-R enterprise schema may define certain constraints to which the contents of a database must conform.

Types of constraints are-

- **mapping cardinalities and**
- **participation constraints**

MAPPING CARDINALITIES-

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

Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

For a binary relationship set *R* between entity sets *A* and *B*,

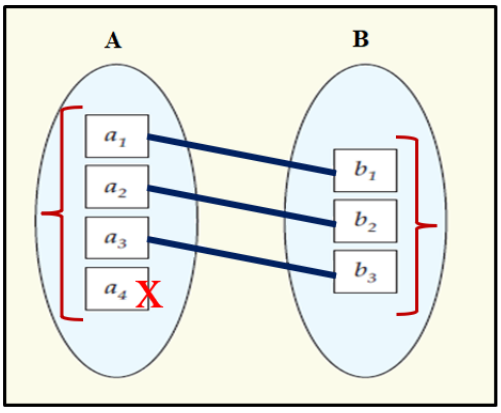
four possibilities are usually specified:

1. one-to-one (1:1)
2. one-to-many (1:N)
3. many-to-one (N:1)
4. many-to-many (M:N)

1. ONE-TO-ONE -

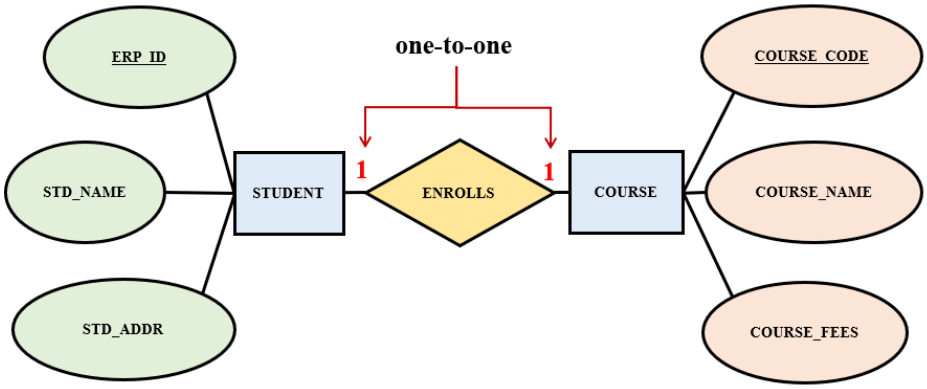
An entity in *A* is associated with *at most* one entity in *B*, and an entity in *B* is associated with *at most* one entity in *A*.

Entities in entity set A
{a1,a2,a3,a4}



Entities in entity set B
{b1,b2,b3}

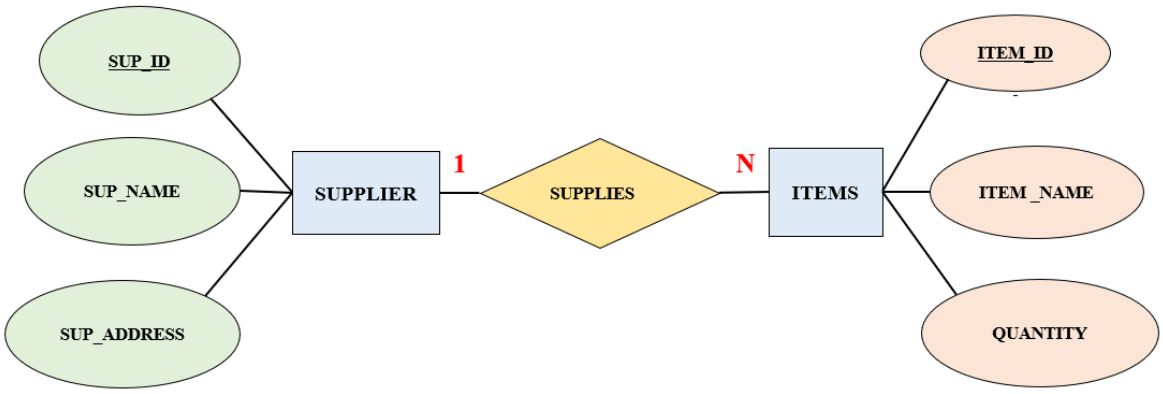
One student can enroll for one course only.



2. **ONE-TO-MANY-**

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

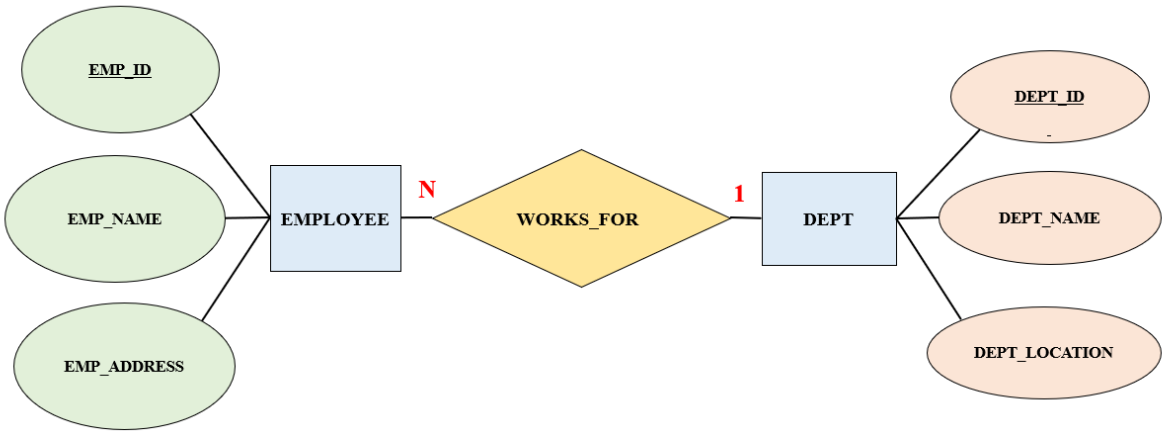
One supplier supplies many (or more than one) items to departmental store.



• **MANY-TO-ONE-**

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

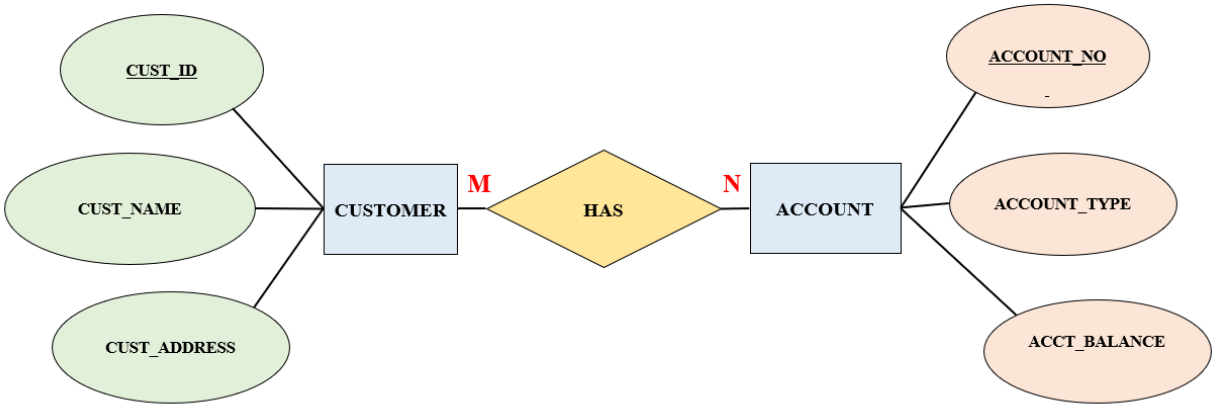
More than one employee is working for one department in company, but only one department is assigned to each employee.



• **MANY-TO-MANY-**

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

One customer has more than on account in bank while one account has many account holders (joint account).



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

PARTICIPATION CONSTRAINTS-

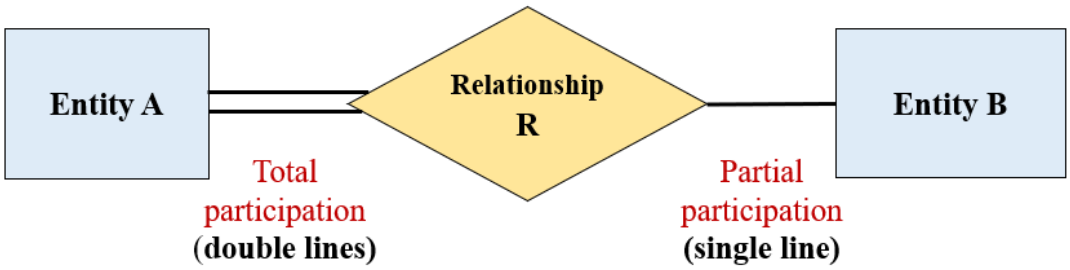
An **entity set** may participate in a relationship either **totally or partially**.

- The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*. Total participation is represented by **double lines**.

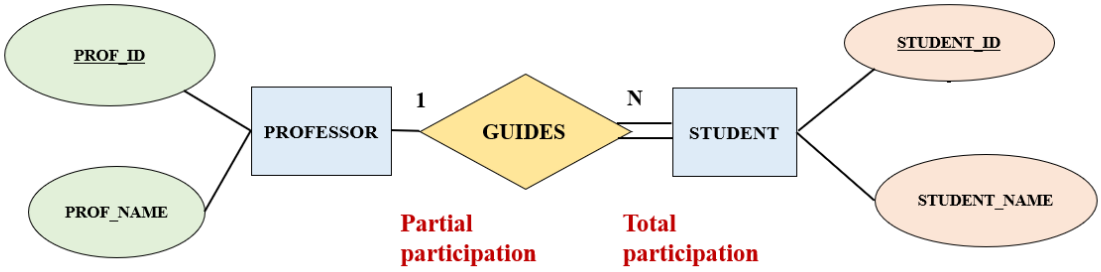
Example- Every student is working on project work.

- If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**. Partial participation is represented by **single line**.

Example- Not all students are taking part in annual sports.



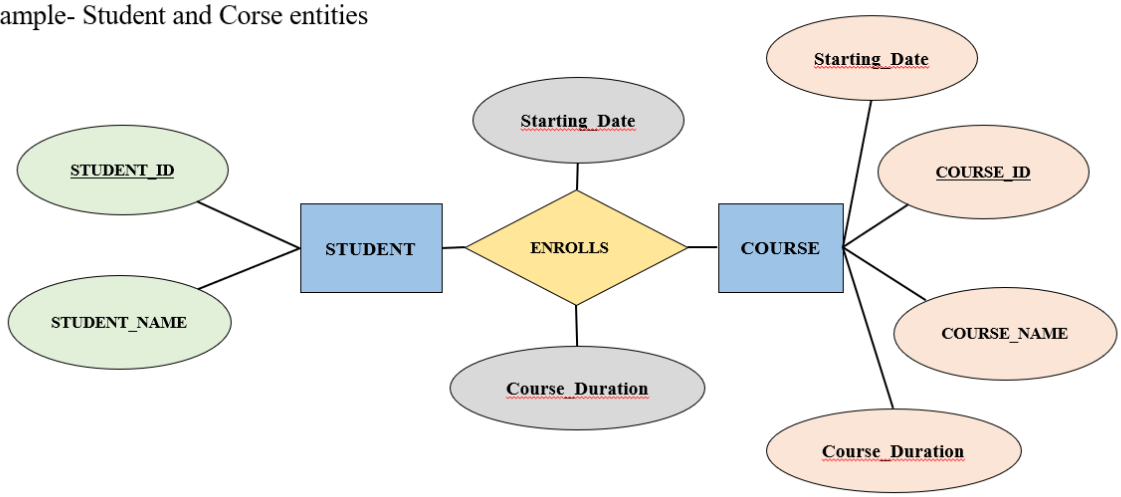
Every **STUDENT** is involved in **GUIDES** relationship by doing the project work while every **PROFESSOR** is not involved in relationship as a project Guide.



Attributes for Relationship Types

In ER model, entities have attributes which can be of various types like single-valued, multi-valued, composite, simple and derived. But relationships can also have attributes associated to them.

Example- Student and Corse entities



KEYS

We must have a way to specify how entities within a given entity set are distinguished.

Conceptually, individual entities are distinct; from a database perspective, however, the differences among them must be expressed in terms of their attributes.

Therefore, the values of the attribute values of an entity must be such that they can *uniquely identify* the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.

The notion of a *key* for a relation schema applies directly to entity sets. That is, a key for an entity is a set of attributes that suffice to distinguish entities from each other. The concepts of superkey, candidate key, and primary key are applicable to entity sets just as they are applicable to relation schemas.

Keys also help to identify relationships uniquely, and thus distinguish relationships from each other. Below, we define the corresponding notions of keys for relationships.

The **primary key** of an entity set allows us to distinguish among the various entities of the set.

ENTITY-RELATIONSHIP DIAGRAMS

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

Basic Structure

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

• **Rectangles divided into two parts** represent entity sets. The first part, which in this textbook is shaded blue, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.

• **Diamonds** represent relationship sets.

• **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.

• **Lines** link entity sets to relationship sets.

• **Dashed lines** link attributes of a relationship set to the relationship set.

• **Double lines** indicate total participation of an entity in a relationship set.

• **Double diamonds** represent identifying relationship sets linked to weak entity sets.

WEAK ENTITY SETS-

An entity set that does not have sufficient attributes to form a primary key is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**. Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**. The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total. The identifying relationship set should not have any descriptive attributes, since any such attributes can instead be associated with the weak entity set.

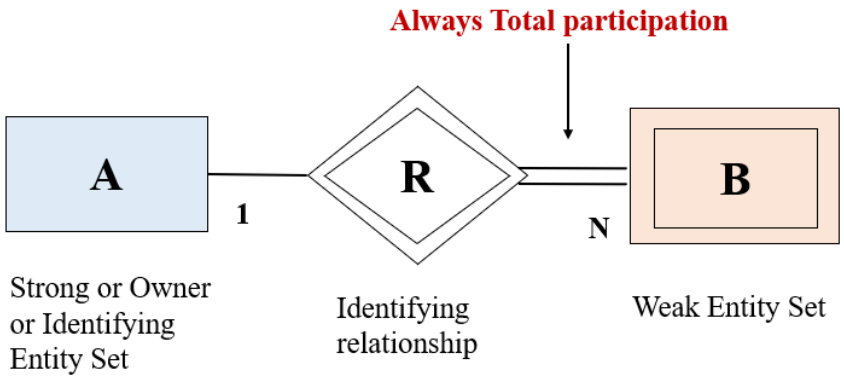
Although a weak entity set does not have a primary key, we nevertheless need a means of distinguishing among all those entities in the weak entity set that depend on one particular strong entity. The **discriminator** of a weak entity set is a set of attributes that allows this distinction to be made. The discriminator of a weak entity set is also called the ***partial key*** **of the entity set.**

The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.

PK of weak entity set = {PK of Identifying entity set} + {Discriminator of Weak entity set}

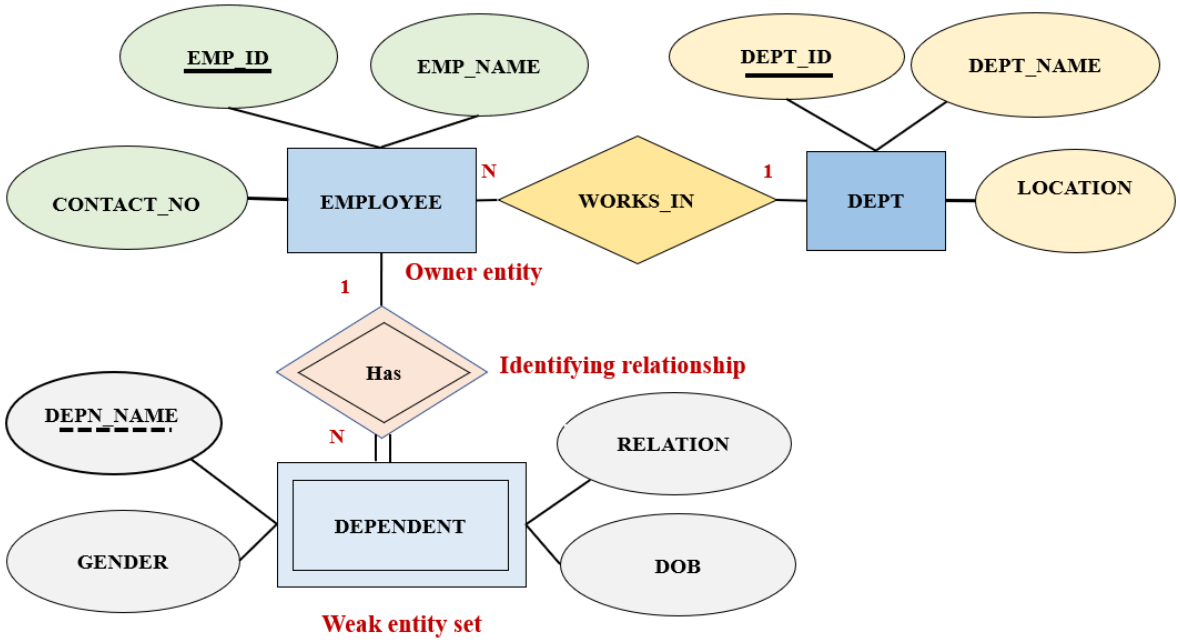
In E-R diagrams, a weak entity set is depicted via a rectangle, like a strong entity set, but there are two main differences:

- The discriminator of a weak entity is underlined with a dashed, rather than a solid, line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.



Example- Draw ER diagram considering following description.

- A company wants to create a database for their employees.
- For every employee detail such as employee ID, name and contact number are stored.
- Every employee works for a department.
- Every department has department ID, name and location.
- A company also wish to store the information about dependents of an employee (such as family members of employee like parents, children, spouse) for an insurance purpose as a social responsibility of a company.
- Company may keep the record of dependent's name, relation with employee, DOB and gender.



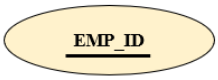
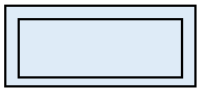
EMPLOYEE

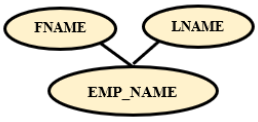
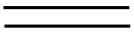
EMP_ID	EMP_NAME	CONTACT_NO
E101	Rahul	111111111
E109	Sachin	22222222
E125	Virendra	33333333

DEPENDENT

EMP_ID	DEPN_NAME	RELATION	GENDER	DOB
E101	Aarati	Wife	F	
E101	Akash	Son	M	
E101	Anuradha	Daughter	F	

ENTITY-RELATIONSHIP DIAGRAM NOTATIONS/SYMBOLS

Rectangle	Strong Entity	
Oval	Attribute of entity	
Diamond	Relationship between Entity Sets	
Line	Link between entity set & attribute and entity set & relationship	
Underlined Attribute	Primary key attribute	
Double sided rectangle	Weak entity set	

Dotted oval	Derived Attribute	
Oval	Multi valued attribute	
Oval linked to oval	Composite Attribute	
Double sided diamond	Weak entity set relationship	
Single line between entity & relationship	Partial participation	
Double line between entity & relationship	Total Participation	
Attribute with dotted underline	Discriminator of weak entity set	

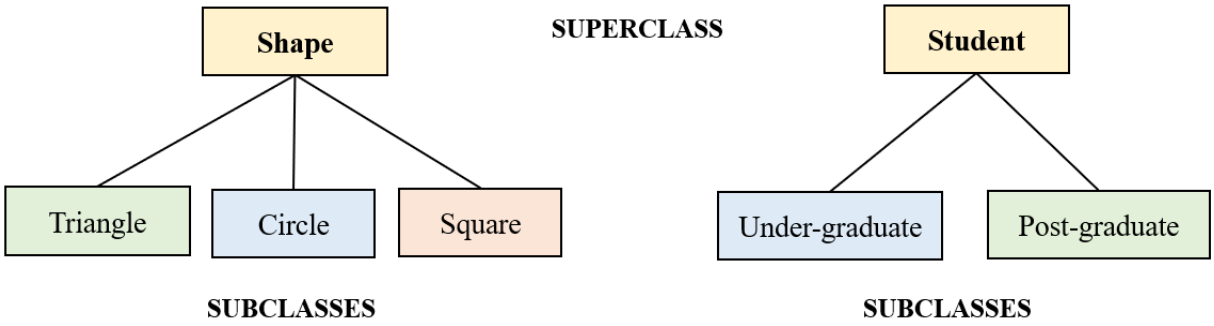
EXTENDED E-R FEATURES IN EER MODEL

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model.

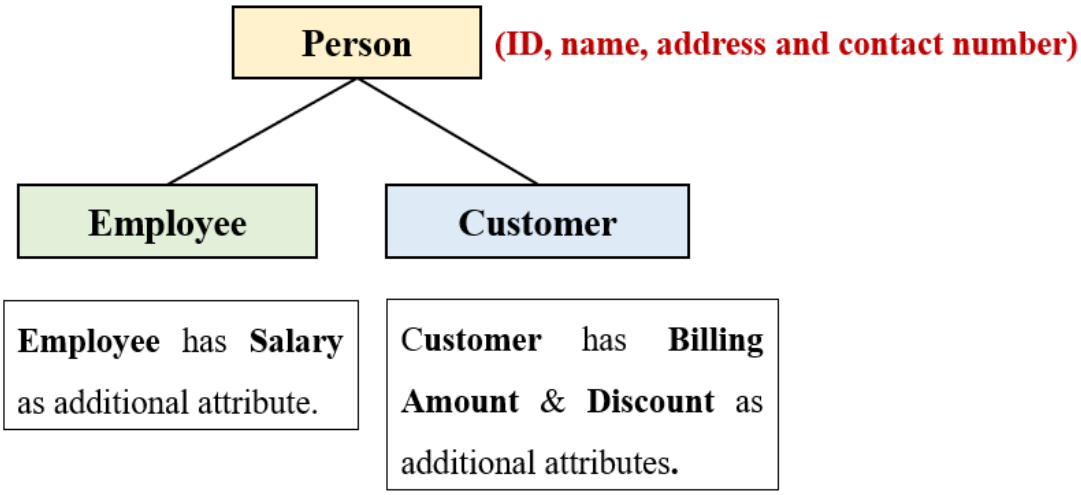
In addition to ER model concepts, **extended features** of E-R (EER) model are:

1. higher- and lower-level (subclasses and super classes) entity sets
2. attribute inheritance
3. specialization
4. generalization and
5. aggregation

1. SUPERCLASS AND SUBCLASS



2. ATTRIBUTE INHERITANCE



3. SPECIALIZATION

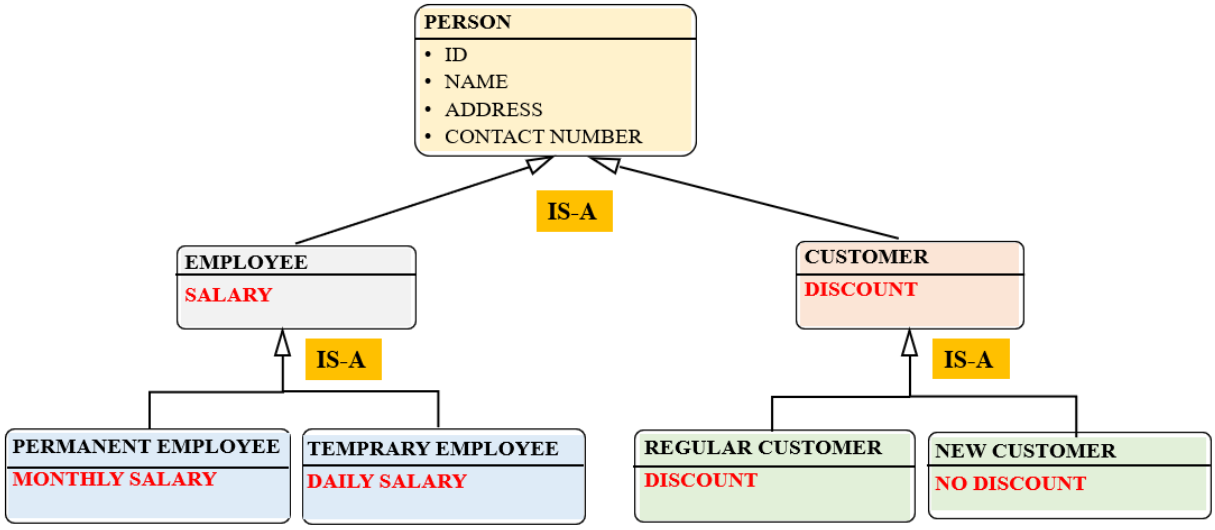
An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings.

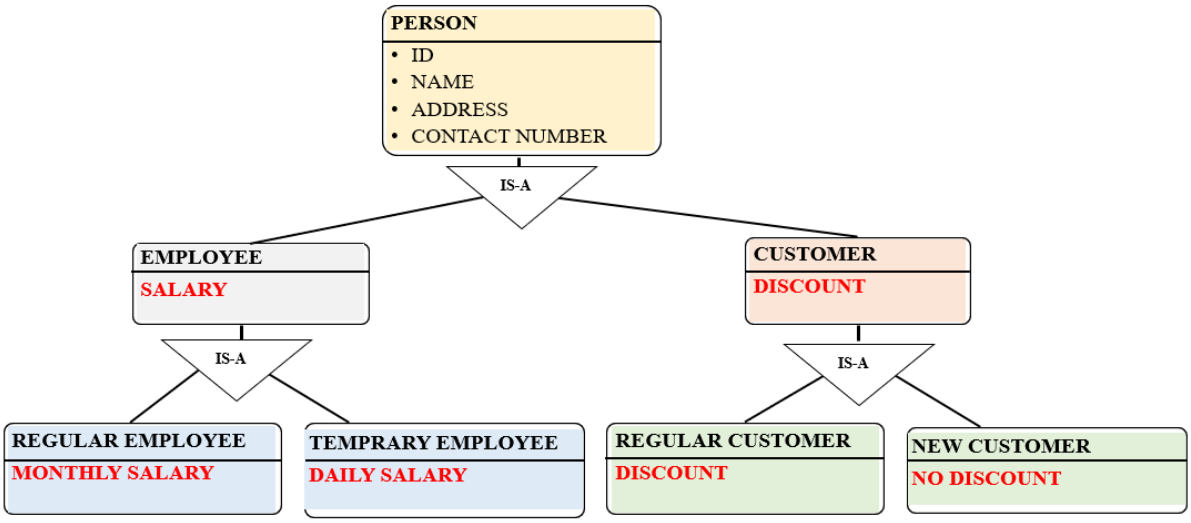
As an example, the entity set *person* may be further classified as one of the following:

• *employee*.
• *student*.

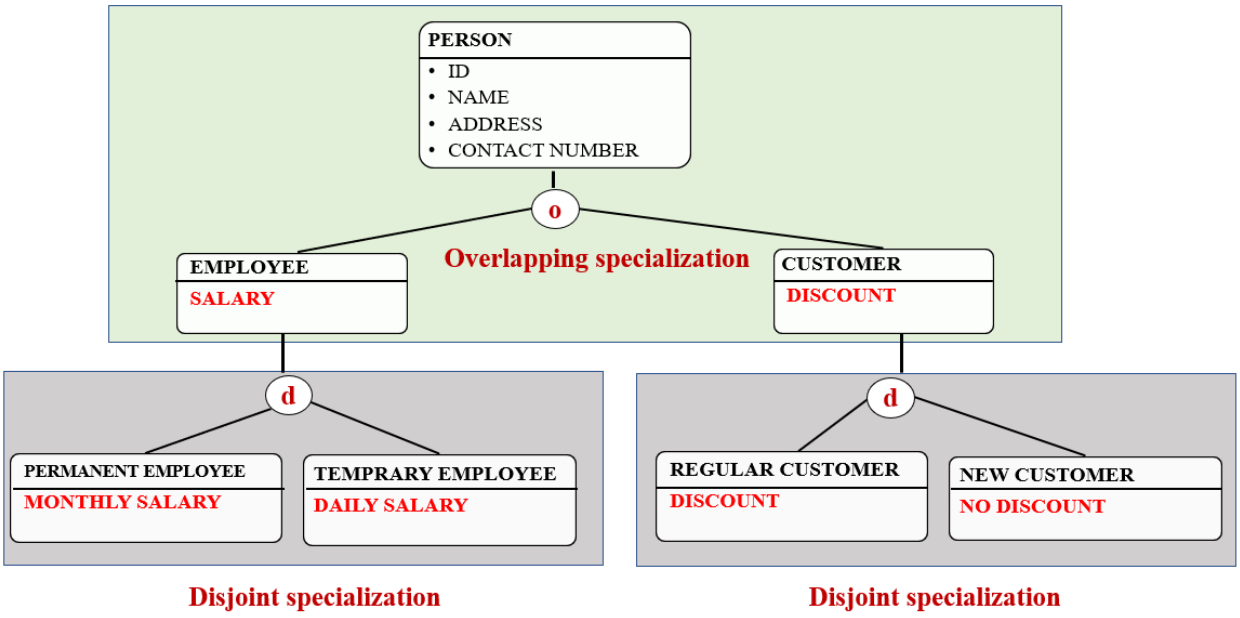
Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For example, *employee* entities may be described further by the attribute *salary*, whereas *student* entities may be described further by the attribute *tot cred*. The process of designating subgroupings within an entity set is called **specialization**. The specialization of *person* allows us to distinguish among person entities according to whether they correspond to employees or students: in general, a person could be
an employee, a student, both, or neither.

As another example, suppose the university divides students into two categories: graduate and undergraduate. Graduate students have an office assigned to them. Undergraduate students are assigned to a residential college. Each of these student types is described by a set of attributes that includes all the attributes of the entity set *student* plus additional attributes.
The university could create two specializations of *student*, namely *graduate* and *undergraduate*. As we saw earlier, student entities are described by the attributes *ID*, *name*, *address*, and *tot cred*. The entity set *graduate* would have all the attributes of *student* and an additional attribute *office number*. The entity set *undergraduate* would have all the attributes of *student*, and an additional attribute *residential college*.
An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs. Another, coexistent, specialization could be based on whether the person is a temporary (limited term) employee or a permanent employee, resulting in the entity sets *temporary employee* and *permanent employee*. When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations. For instance, a given employee may be a temporary employee who is a secretary.
In terms of an E-R diagram, specialization is depicted by a hollow arrow-head pointing from the specialized entity to the other entity.
We refer to this relationship as the ISA relationship, which stands for “is a” and represents, for example, that an instructor “is a” employee.





The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**.



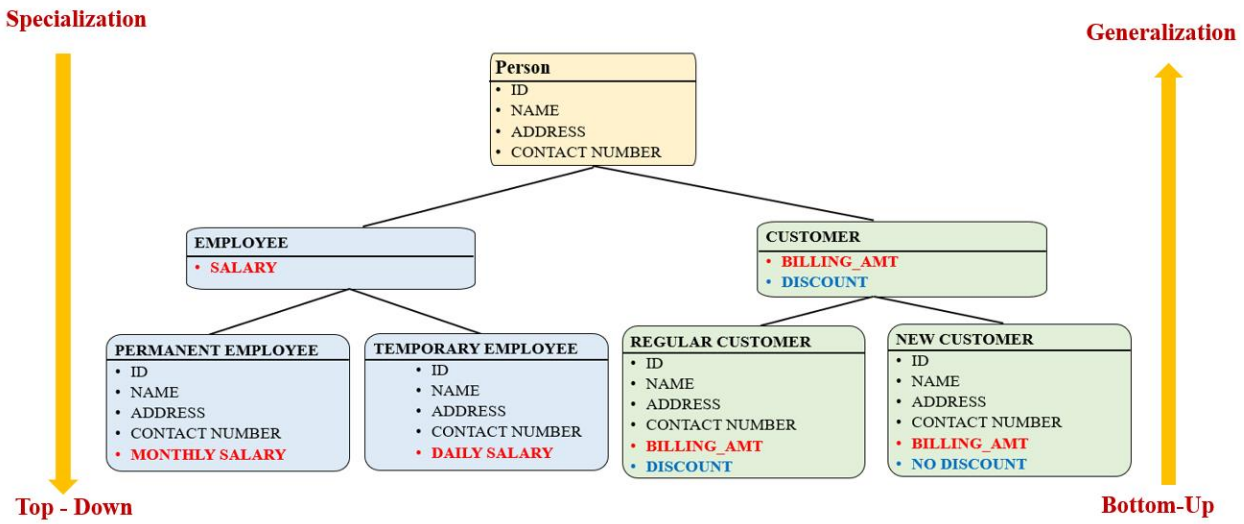
4. GENERALIZATION

The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit.

The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features.

Generalization is the design process in a **bottom-up manner**, in which multiple entity sets are **grouped into a higher-level entity set** on the basis of **common/generalized attributes**.

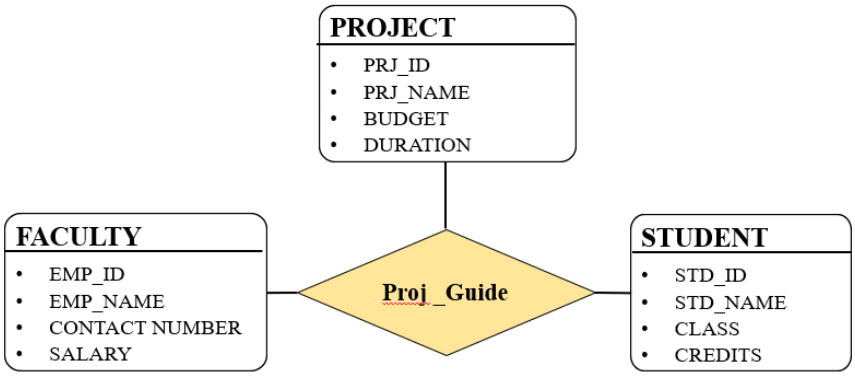
Generalization, is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets.



For all practical purposes, generalization is a simple inversion of specialization. We apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation are distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

5. AGGREGATION

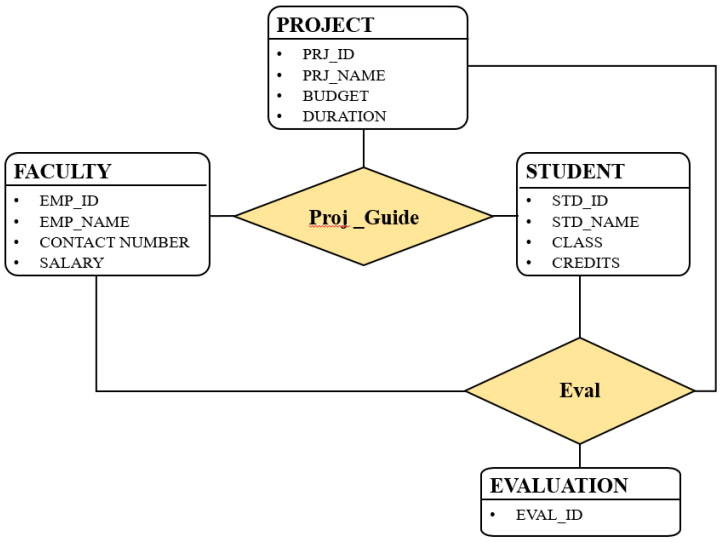
One limitation of the E-R model is that it cannot express relationships among relationships.
Consider the ternary relationship **Proj_Guide**, between an **FACULTY, STUDENT** and **PROJECT** entity sets as,



Now suppose that each faculty guiding a student on a project is required to **prepare a monthly evaluation report.**

We model the evaluation report as an entity **EVALUATION,** with a primary key **Evaluation ID**.

One alternative for recording the **(STUDENT, PROJECT, FACULTY)** combination to which an **EVALUATION** corresponds is to create a quaternary (4-way) relationship set **Eval_for** between **FACULTY, STUDENT, PROJECT** and **EVALUATION**.



Relationship sets **Proj_Guide** and **Eval_for may be combined** into one single relationship set. But **can not be combined** them into a single relationship, since some faculty, student, project combinations **may not have an associated evaluation.**

There is redundant information in the resultant figure, however, since every **FACULTY, STUDENT, PROJECT** combination in **Eval** must also be in **Proj_Guide**.

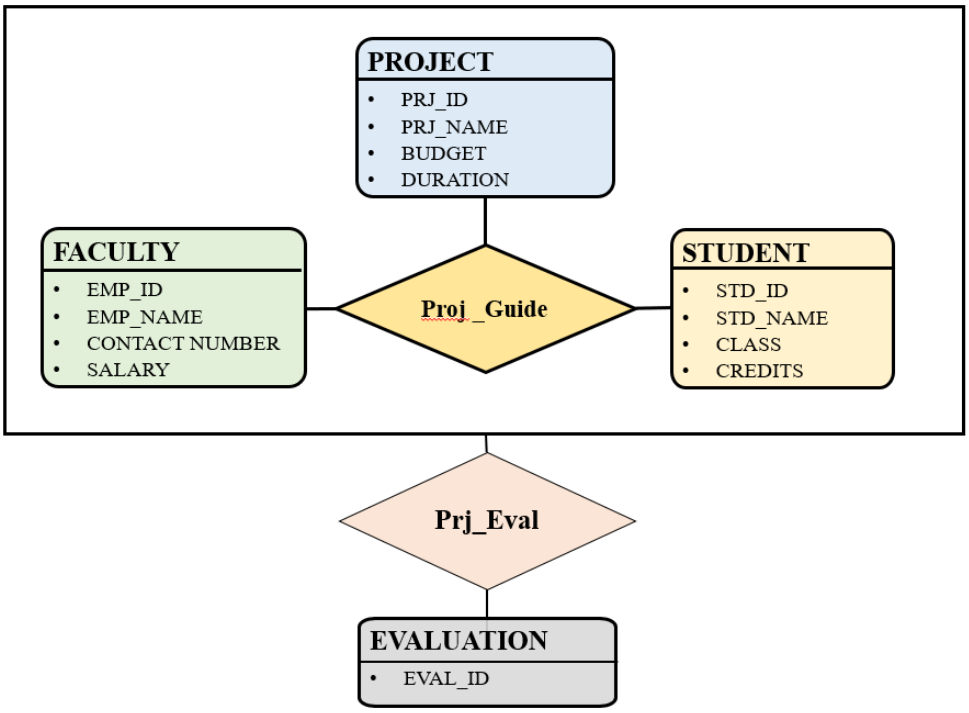
The best way to model a situation is to use **aggregation**.

Aggregation is an abstraction through which **relationships between entities are treated as higher-level entities**.

For example,

we regard the relationship set **Proj_Guide** (relating the entity sets **FACULTY, STUDENT** and **PROJECT**) as a higher-level entity set called **Proj_Guide.** Such an entity set is treated in the same manner as is any other entity set.

We can then create a binary relationship **Prj_Eval** between **Proj_Guide** and **EVALUATION** to represent which (**STUDENT, PROJECT, FACULTY**) combination an evaluation is for.



Reference: Database System Concepts, Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Sixth Edition Mcgraw-Hill Publication
(Chapter-7 Korth- 6th Edition)

Chapter 3

RELATIONAL MODEL AND RELATIONAL ALGEBRA

INTRODUCTION TO THE RELATIONAL MODEL

Relational Model is Proposed by Edgar. F. Codd.

Relational Model is a primary data model for commercial data processing applications.

It is a simple and elegant model with a mathematical basis and set theory.

Most of the modern DBMS are relational databases.

Relational algebra operations plays crucial role in query optimization and execution.

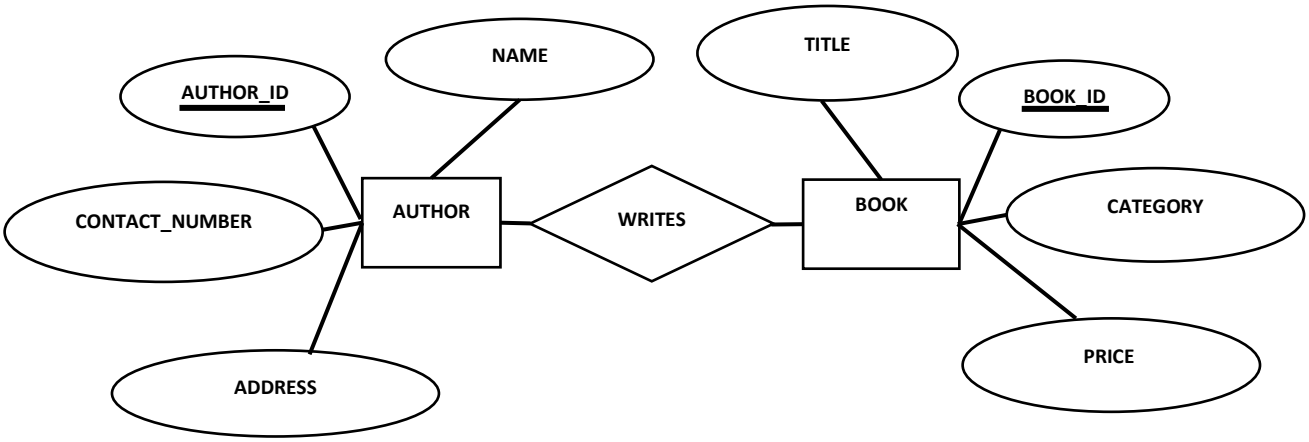
STRUCTURE OF RELATIONAL DATABASES

A relational database consists of collection of tables and each table has unique name.

In the relational model, the term

- relation (mathematical concept) is used to refer to a table,
- tuple (a sequence (or list) of values) is used to refer to a row/record and
- attribute refers to a column of a table.

Example- consider an ER diagram as



The above ER diagram can be represented in form of tables as

AUTHOR:

AUTHOR_ID	NAME	CONTACT_NUMBER	ADDRESS
101	Dan Brown	1234567890	USA
102	Ruskin Bond	3456789012	India
103	R. K. Narayan	5678901234	India
104	J. K. Rowling	7890123456	England

BOOK:

BOOK_ID	TITLE	CATEGORY	PRICE
B1	The Da Vinci Code	Science Fiction	450
B2	Room on the Roof	Novel	250
B3	Malgudi Days	Short stories	350
B4	Harry Potter	Fiction	550

In general, a row in a table represents a relationship among a set of values.

RELATION INSTANCE

The term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows.

For example, instance of the relation **AUTHOR** has 4 tuples corresponding to 4 authors and instance of the relation **BOOK** has 4 tuples corresponding to 4 books.

The order in which tuples appear in a relation is irrelevant, since a relation is a **set** of tuples.

Domain of attribute

For each attribute of a relation, there is a **set of permitted values**, called the **domain** of that attribute.

Thus, the domain of the **Salary** attribute of the **Faculty** relation is the set of all possible salary values, while the domain of the **Name** attribute is the set of all possible faculty names.

Attribute values are (normally) required to be **atomic**; that is, **indivisible.**

Some attribute values may be null values. The **null** value is a special value that signifies that the value is unknown or does not exist.

For example, suppose the attribute **CONTACT_NUMBER** in the **AUTHOR** relation.

It may be possible that author does not have a phone number at all, or that the contact number is unlisted. We would then have to use the **null value** to signify that the value is **unknown or does not exist.**

RELATION SCHEMA

A relation schema consists of a **relation name, list of attributes or column names and their corresponding domains.**

If $\mathbf{A_1, A_2, \ldots, A_n}$ are attributes of relation **R**, then

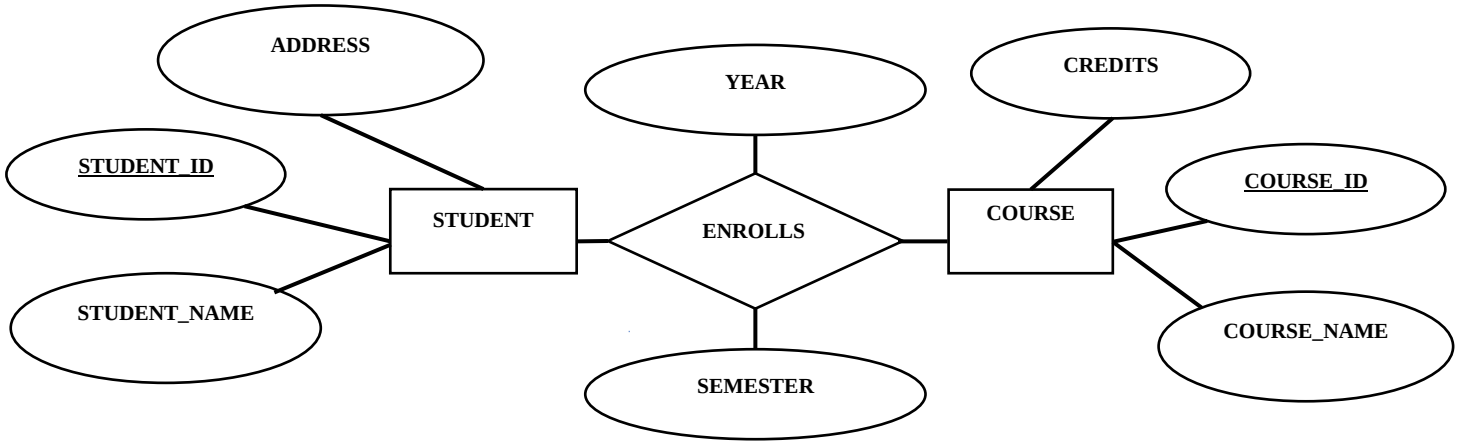
$\mathbf{R = (A_1, A_2, \ldots, A_n)}$ is a relation schema.

Example:

AUTHOR = (Author_ID, Name, Contact_Number, Address)

BOOK = (Book_ID, Title, Category, Price)

Example- Relation schema



STUDENT = (STUDENT_ID, STUDENT_NAME, ADDRESS)

COURSE = (COURSE_ID, COURSE_NAME, CREDITS)

ENROLLS = (STUDENT_ID, COURSE_ID, YEAR, SEMESTER)

DATABASE SCHEMA & INSTANCE

Database schema is the logical structure of the database.

Example:

- AUTHOR = (Author_ID, Name, Contact_Number, Address)
- BOOK = (Book_ID, Title, Category, Price)

Database instance is a snapshot of the data in the database at a given instant in time.

For relations AUTHOR and BOOK database instance can be

AUTHOR_ID	NAME	CONTACT_NUMBER	ADDRESS
101	Dan Brown	1234567890	USA
102	Ruskin Bond	3456789012	India
103	R. K. Narayan	5678901234	India
104	J. K. Rowling	7890123456	England

BOOK_ID	TITLE	CATEGORY	PRICE
B1	The Da Vinci Code	Science Fiction	450
B2	Room on the Roof	Novel	250
B3	Malgudi Days	Short stories	350
B4	Harry Potter	Fiction	550

KEYS IN RELATIONAL MODEL

There must be a way to specify **how tuples** within a given relation are **distinguished**.

Key is a **set of attributes of relation** whose values **uniquely identify** a **tuple** in any instance.

In other words, **no two tuples in a relation** are allowed to have **exactly the same value** for all attributes.

There are **four types of keys** in relational model

- **Super key**
- **Candidate key**
- **Primary key**
- **Foreign key**

SUPER KEY

A **super key** 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 STUDENT is sufficient to distinguish one student tuple from another.

Thus, ID is a super key. The name attribute of student, on the other hand, is not a super key, because several student smight have the same name.

A super key **may contain extraneous attributes**.

Example: STUDENT = (STUDENT_ID, NAME,ADDRESS)

For **STUDENT** entity set, **super key** can be

- **{STUDENT_ID}**
- **{STUDENT_ID, NAME}**
- **{STUDENT_ID, NAME, ADDRESS}**

CANDIDATE KEY

Minimal or subset of super keys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key.

Example:

STUDENT = (STUDENT_ID, NAME,ADDRESS)

For **STUDENT** entity set, **candidate key** can be

- **{STUDENT_ID}**
- **{STUDENT_ID, NAME}**

PRIMARY KEY

Primary key is a **candidate key that is chosen by the database designer** as the principal **means of identifying tuples** within a relation.

Examples:

- **STUDENT =** (STUDENT_ID, STUDENT_NAME,ADDRESS)

  STUDENT_ID s primary key.
- **AUTHOR =** (AUTHOR_ID, Name, Contact_Number, Address)

  AUTHOR_ID is primary key.
- **BOOK =** (BOOK_ID, Title, Category, Price)

  BOOK_ID is primary key.

FOREIGN KEY

A relation (B) **may include** among its attributes the **primary key of another relation** (A). This attribute is called a **foreign key** from relation B referencing relation A.

The relation **B** is also called the **referencing relation** of the foreign key dependency, and relation **A** is called the **referenced relation** of the foreign key.

Example:

Consider two relations, EMPLOYEE and PROJECT having relationship Works_on.

Keys -Example

EMPLOYEE relation

EMP_ID	FNAME	MNAME	LNAME	CONTACT_NUMBER	ADDRESS
101	Sachin	R	Tendulkar	123456789	Vashi
102	Rahul	S	Dravid	234567891	Thane
103	Anil	R	Kumble	345678912	Panvel
104	Virat	P	Kohli	456789123	Andheri

PROJECT relation

PRJ_ID	PRJ_NAME	CATEGORY	EMP_ID
P1	Home Automation	IOT	101
P2	Leave Management	Database	102
P3	Network Setup	Networking	103
P4	Online sales Management	Web App	104

Super-key for relation EMPLOYEE can be

=(EMP_ID,FNAME,MNAME,LNAME,CONTACTNUMBER,ADDRESS)

Candidate key for relation EMPLOYEE can be

=(EMP_ID,FNAME,MNAME,LNAME)

=(FNAME,MNAME,LNAME,CONTACTNUMBER)

Primary key

for relation EMPLOYEE is EMP_ID and

for relation PROJECT is PRJ_ID

Foreign key

EMP_ID is a foreign key for PROJECT relation.

The relation **PROJECT** is called the **referencing relation** and relation EMPLOYEE is called the **referenced relation** of the foreign key.

ASPECTS OF KEYS IN RELATIONAL MODEL

- A key (whether primary, candidate, or super) is a property of the **entire relation**, rather than of the individual tuples.
- Any two individual tuples in the relation are prohibited from having the **same value on the key attributes** at the same time.
- The key represents a constraint in the real-world enterprise being modelled.
- There can be **more than one foreign key in a relation** scheme.
- The primary key should be chosen such that its attribute values are **never, or very rarely, changed.**

  For example, Aadhar (UID) numbers in India are guaranteed never to change.
- **Unique identifiers** generated by enterprises generally do not change, like **EMP_ID**.
- It is customary to list the primary key attributes of a relation schema before the other attributes.
    - **Emp_ID** attribute of Employee is l**isted first**, since it is the primary key.
    - Primary key attributes are also **underlined**.
- Attributes which are **likely to change its value** should not be part of the primary key,
    - **Example: Address** of a person
- Attributes which are **likely to have same value** should not be part of the primary key.
    - **Example : Name** of a person

INTEGRITY CONSTRAINTS IN RELATIONAL MODEL

Integrity constraints are **necessary conditions to be satisfied by the data values** in the relational instances so that the set of data values constitute a **meaningful database**.

TYPES OF INTEGRITY CONSTRAINTS:

- **Domain Constraints**
  - Actual values of an attribute in any tuple must belong to the declared domain i.e. values must from set of **permissible values**.
- **Key Constraints**
  - Two tuples in any relation instance **should not have identical values** for attributes.
  - Key attributes (Super, candidate & primary) **cannot have null value**.
- **Foreign Key Constraints**
  - Value in one relation appear in another relation (relation can have **foreign key/s**).

REFERENTIAL INTEGRITY CONSTRAINT

Referential integrity requires that a **foreign key** must have

- a **matching primary key** or
- a **null** value.

This **constraint** is specified between two tables (parent and child); it maintains the **correspondence** between rows in these tables.

It means the **reference** from a row in one table to another table **must be valid**.

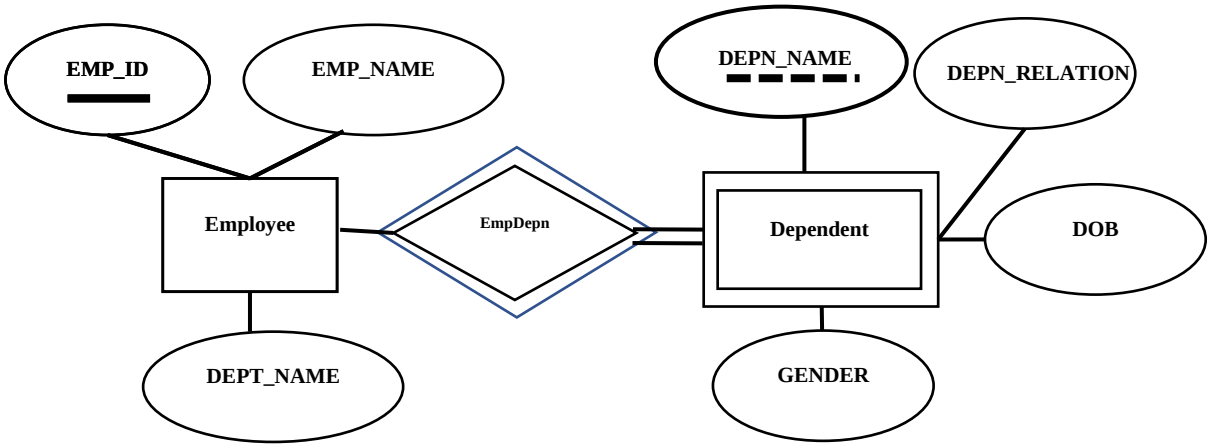
Conditions for referential integrity

- The related fields have the **same data type**.
- Both **tables** belong to the **same database**.

MAPPING THE ER AND EER MODEL TO THE RELATIONAL MODEL

- A database which conforms to an **E-R diagram** can be represented by a **collection of tables**.
- For each entity set and relationship set there is a **unique table** which is assigned the **name** of the corresponding entity set or relationship set.
- Each table has a number of **columns** (generally corresponding to attributes), which have **unique names**.
- **Primary keys** allow **entity sets and relationship sets** to be expressed uniformly as tables which represent the contents of the database.

Example



REPRESENTING STRONG ENTITY SETS AS TABLES

A **strong entity set** reduces to a table with the same attributes.

EMP_ID	EMP_NAME	DEPT_NAME
101	Sachin Tendulkar	Human Resource
102	Rahul Dravid	Accounts

REPRESENTING WEAK ENTITY SETS

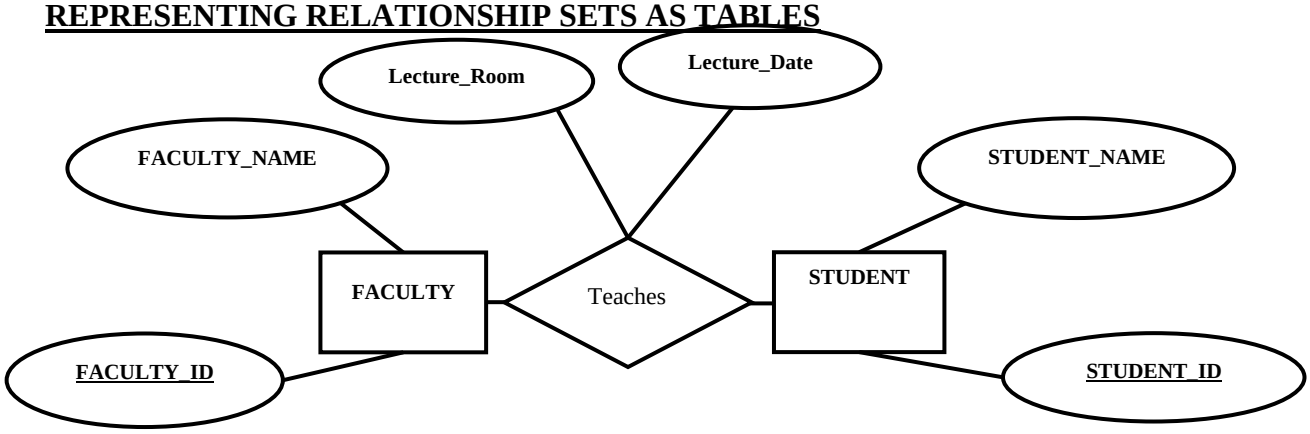
A **weak entity set** becomes a table that includes a column for the primary key of the identifying strong entity set.

EMP_ID	DEP_NAME	DEP_RELATION	GENDER
101	Arjun	Son	Male
101	Sara	Daughter	Female
102	Samit	Son	Male

Composite primary key = (EMP_ID, DEP_NAME)

EMP_ID is primary key of identifying strong entity set (Employee) while DEP_NAME is discriminator of weak entity set.

REPRESENTING RELATIONSHIP SETS AS TABLES



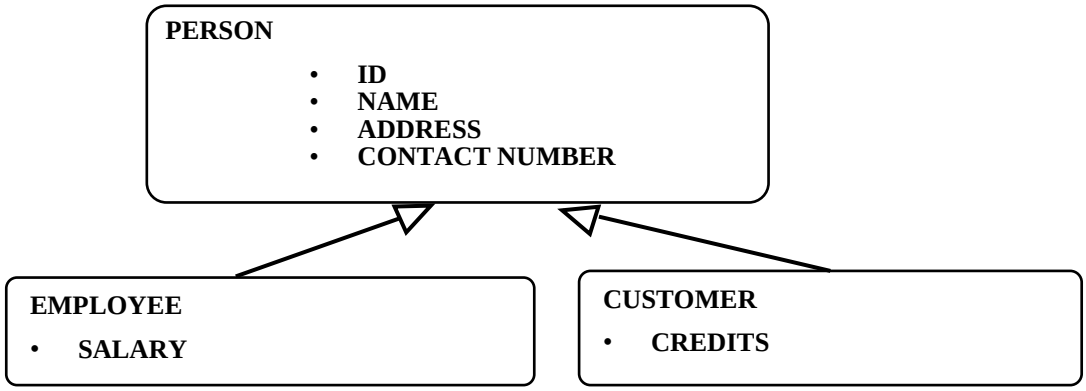
Teaches

FACULTY_ID	STDUENT_ID	LECTURE_DATE	LECTURE_ROOM
101	S1201	--/--/----	C-608
109	S1301	--/--/----	C-505

REPRESENTING SPECIALIZATION AS TABLES

METHOD 1:

- Form a table for the generalized entity account
- Form a table for each entity set that is generalized (include primary key of generalized entity set)



PERSON

ID	NAME	ADDRESS	CONTACT_NUMBER

EMPLOYEE

ID	SALARY

CUSTOMER

ID	CREDITS

METHOD 2:

- Form a table for each entity set that is generalized.

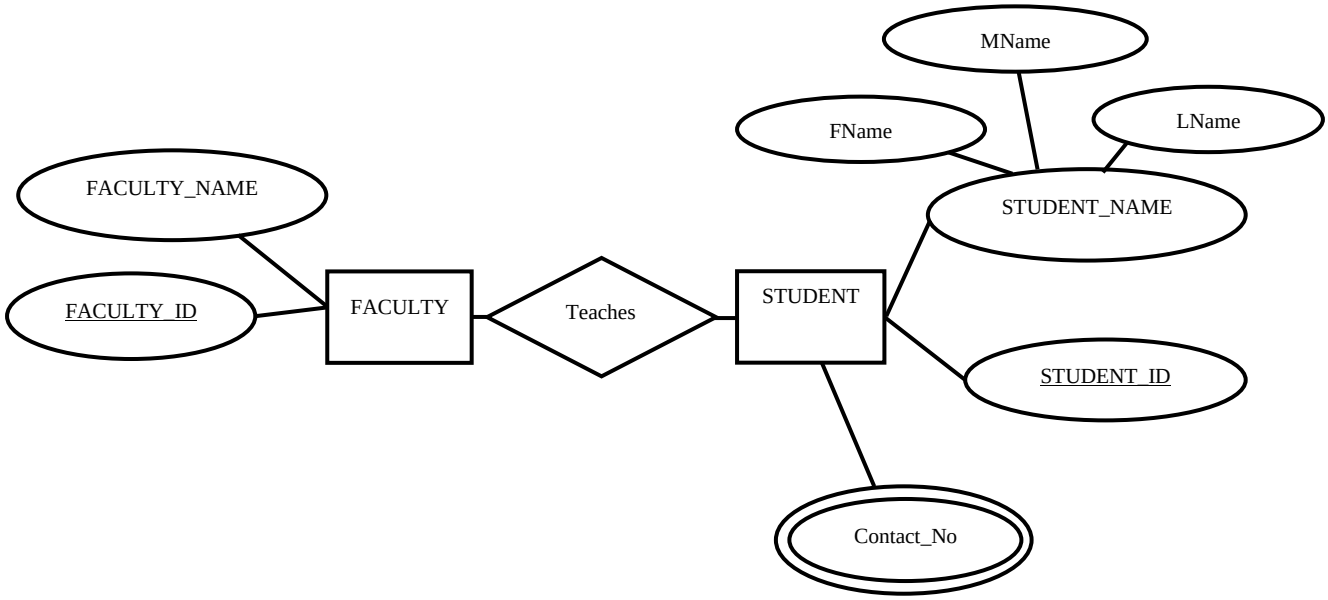
EMPLOYEE

ID	NAME	ADDRESS	CONTACT_NUMBER	SALARY

CUSTOMER

ID	NAME	ADDRESS	CONTACT_NUMBER	CREDITS

COMPOSITE AND MULTIVALUED ATTRIBUTES TO TABLES



COMPOSITE ATTRIBUTES

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

Example: for entity set **STUDENT** with composite attribute **Name** with component attributes **first-name, middle name** and **last-name** the table corresponding to the entity set can be

MULTIVALUED ATTRIBUTES

- A multivalued attribute M of an entity E is represented by a separate table EM.
- Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M.
- Each value of the multivalued attribute maps to a **separate row of the table** EM.
- Example: **CONTACT_NUMBER** of entity set **STUDENT**

STUDENT

STUDENT_ID	CONTACT_NUMBER
201	1234567890
201	2345678901
301	3456789012
701	4567890123
701	5678901234

RELATIONAL ALGEBRA (RA)

- The relational algebra is a **procedural query language** that means it gives a **procedural method of specifying a query** for retrieval of data.
- It consists of a **set of operators (unary and binary)** that take (one or two) **relation instances as input** (arguments) and return **new relations as their result**.
- SQL queries are **internally translated** into **relational algebra** expressions.
- It forms the **core component** of a **relational query engine**.
- It provides a **framework** for **query optimization**.

OPERATIONS IN RELATIONAL ALGEBRA

The **six fundamental operations** in the relational algebra are

- **Select (σ) (sigma)**
- **Project (∏) (Pi)**
- **Union (U)**
- **Set difference (–)**
- **Cartesian product (X)**
- **Rename (ρ) (rho)**

In addition to the fundamental operations, there are several other operations—namely,

- **set intersection**
- **Natural join**
- **assignment**

UNARY and BINARY types of operations

Unary operations which operate on one relation are

- **Select (σ)**
- **Project (∏)**
- **Rename (ρ)**

Binary operations which operate on pairs of relations are

- **Union (U)**
- **Set difference (–)**
- **Cartesian product (X)**

SELECT OPERATION (σ)

The **select** operation **selects tuples that satisfy a given predicate**.

Notation: $\sigma_p(r)$

- The lowercase Greek letter **sigma (σ)** denotes the **selection**.
- The **selection predicate *p*** appears as a **subscript to** σ.
- The **argument relation *r*** is in **parentheses** after the σ.

Example- Select Operation

AUTHOR

AUTHOR_ID	NAME	CONTACT_NUMBER	ADDRESS
101	Dan Brown	1234567890	USA
102	Ruskin Bond	3456789012	India
103	R. K. Narayan	5678901234	India
104	J. K. Rowling	7890123456	England

Example: select those tuples of the AUTHOR relation where the author name is Ruskin Bond.

σ Name = "Ruskin Bond" (AUTHOR)

AUTHOR_ID	NAME	CONTACT_NUMBER	ADDRESS
102	Ruskin Bond	3456789012	India

SELECT OPERATION WITH COMPARISON OPERATORS AND CONNECTIVES

In general, comparisons can be done using **=, ≠, <, ≤, >,** and **≥ operators** in the selection predicate.

Furthermore, several predicates can be also combined into a larger predicate by using the connectives **and (∧), or (∨),** and **not (¬).**

Select operation is **commutative**:

σc1 (σc2(*r*)) = σc2 (σc1(*r*))

Examples:

1. To find the employees whose salary is more than 60000

σ SALARY > 60000 **(EMPLOYEE)**

2. To find the employees whose salary is more than 50000 and works in Testing department.

σ SALARY >50000 ^ DEPT = "Testing" (**EMPLOYEE**)

3. Obtain information about employees whose salary is between 40000 and 50000

σ SALARY ≥40000 ^ SALARY < 50000 (**EMPLOYEE**)

PROJECT OPERATION (∏)

Project Operation is a unary operation that **returns its argument relation**, with **certain attributes left out.**

Projection is denoted by the uppercase Greek letter **pi (∏).**

We list those attributes that we wish **to appear in the result** as a subscript to ∏ .

Notation: $\prod_{A1,A2,A3 \ldots Ak} (r)$

where A_1, A_2, …, A_k are attribute names and r is a relation name.

The result is defined as the **relation of k columns** obtained **by removing the columns that are not listed.**

Duplicate rows are removed from result, since relations are sets.

Example- Project Operation

EMPLOYEE

EMP_ID	NAME	SALARY	DEPT
101	Sachin Tendulkar	45000	HR
102	Rahul Dravid	75000	Testing
103	Anil Kumble	55000	Testing
104	Virat Kohli	70000	Marketing

Example: Display ID and names of all employees.

$\prod_{EMP\_ID, NAME}$ **(EMPLOYEE)**

EMP_ID	NAME
101	Sachin Tendulkar
102	Rahul Dravid
103	Anil Kumble
104	Virat Kohli

COMPOSITION OF RELATIONAL OPERATIONS

The **result of a relational-algebra operation** is **relation** and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.

Example: Find the ID and names employees whose salary is more than 60000.

R1 ⟵ σ SALARY > 60000 (EMPLOYEE)

R2 ⟵ ∏ EMP_ID,NAME (R1)

or it can be represented as

∏ EMP_ID,NAME (σ SALARY > 60000 **(EMPLOYEE))**

The output will be

EMP_ID	NAME
102	Rahul Dravid
104	Virat Kohli

UNION OPERATION (U)

The union operation allows us to **combine two relations**. It includes all tuples that are in tables **R or in S**. It also **eliminates duplicate tuples**.

Notation: R U S

For **R** U **S to be valid,**

1. relations R and S must have the **same number of attribute**
2. the attribute **domains** must be **compatible.**

That is,the domains of the ith attribute of R and the ith attribute of S must be the same,for all i.

Example: 2nd column of R deals with the same type of values as does the 2nd column of S

Example- Training institute is conducting different courses-Batch-1 relation stores data of regular courses while Batch-2 relation stores data of week-end courses,

Batch-1

Course_Id	Course_Name
11	Foundation of C
21	C++
31	JAVA

Batch-2

Course_Id	Course_Name
91	Python
21	C++

Batch-1 U **Batch-2**

Course_Id	Course_Name
11	Foundation of C
21	C++
31	JAVA
91	Python

SET OPERATORS ON RELATIONS

SET-INTERSECTION OPERATION (∩)

The set-intersection operation allows us to find tuples that are in both the input relations.

Notation: R ∩ S

For **R ∩ S to be valid,**

1. relations R and S must have the **same number of attributes**.
2. The attribute **domains** must be **compatible.**

Example:

Finds all theory courses taught in both, Odd 2019 semester and in the Even 2020 semester.

∏ Theory_Course_Name (σ semester="Odd" Λ year=2019 (CLASS)) ∩

∏ Theory_Course_Name (σ semester="Even" Λ year=2020 (CLASS))

Theory_Course_Name
Applied Mathematics

SET DIFFERENCE OPERATION (–)

The set-difference operation allows us to find tuples that are in one relation but are not in another.

Notation: R – S

This expression produces a relation containing those tuples in R but not in S.

For **R – S to be valid,**

1. relations R and S must have the **same number of attributes**.
2. The attribute **domains** must be **compatible.**

Example- to find names of students who attended the regular class but not the extra class, then,

∏ Student_Name (RegularClass) - ∏ Student_Name(ExtraClass)

ASSIGNMENT OPERATION (←)

It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.

The assignment operation is denoted by ← and works like assignment in a programming language.

Example: Find all faculty in the "COMPUTER" and "EXTC" department.

Computer ← σ dept_name = "COMPUTER" (FACULTY)

EXTC ← σ dept_name = "EXTC" (FACULTY)

Computer ∪ EXTC

RENAME OPERATION (ρ)

The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ, is provided for that purpose like

ρ (Relation2, Relation1)

To rename STUDENT_SE relation to STUDENT_TE, we can use rename operator like:

ρ (STUDENT_TE, STUDENT_SE)

CARTESIAN-PRODUCT OPERATION (X) OR (CROSS PRODUCT)

The Cartesian-product operation allows us to **combine values from any two relations**.

Notation: R X S

The Cartesian product R × S of two relations R, & S is computed by **concatenating each tuple t ∈ R** with **each tuple u ∈ S**.

Example: Student

Std_ID	Std_Name
101	Sachin
102	Saurav
103	Rahul
104	Virat

Sport

Sports_ID	Sports_Name
1	Cricket
2	Football

Student X Sport

Std_ID	Std_Name	Sports_ID	Sports_Name
101	Sachin	1	Cricket
101	Sachin	2	Football
102	Saurav	1	Cricket
102	Saurav	2	Football
103	Rahul	1	Cricket
103	Rahul	2	Football
104	Virat	1	Cricket
104	Virat	2	Football

JOIN OPERATIONS (⋈)

Join operation is essentially a **cartesian product followed by a selection criterion.**

Join operation denoted by ⋈.

JOIN operation also allows joining of related tuples from different relations.

Types of JOIN:

Inner Join:

- Theta join
- Equi join
- Natural join

Outer join:

- Left Outer Join
- Right Outer Join
- Full Outer Join

INNER JOIN

In an inner join, only those tuples that **satisfy the matching criteria are included, while the rest are excluded.**

THETA JOIN:

The general case of JOIN operation is called a **Theta join**. It is denoted by symbol **θ.**

The theta join operation between relations R & S is defined as follows:

$R \bowtie_\theta S$

where **θ** represents a condition.

Example:

$\mathbf{R \bowtie_{R.column\ 2 > S.column\ 2} (S)}$

EQUI JOIN:

When a theta join uses **only equivalence condition**, it becomes a equi join.

$R \bowtie_{R.column\ 2 = S.column\ 2} (S)$

Example:

$\mathbf{Employee \bowtie_{Emp.ID= PrjEmp.ID} Project}$

NATURAL JOIN (⋈)

Natural join **does not use any comparison operator**.

It **does not concatenate** the way a Cartesian product does.

We can perform a Natural Join only if there is **at least one common attribute** that exists between two relations.

In addition, the **attributes must have the same name and domain**.

Natural join acts on those matching attributes where the **values of attributes** in both the relations are same.

If R and S are relations **without any attributes in common**,

that is, $\mathbf{R \cap S = \emptyset}$ (null),

then $\mathbf{R \bowtie S = R \times S}$

Course

Course_ID	Course_Name	Department
CS01	Database	Computer
EE01	BEE	Electrical
ME01	Applied Mechanics	Mechanical

Faculty

Department	Faculty
Computer	A
Electrical	B
Mechanical	C

Course ⋈ Faculty

Department	Course_ID	Course_Name	Faculty
Computer	CS01	Database	A
Electrical	EE01	BEE	B
Mechanical	ME01	Applied Mechanics	C

OUTER JOINS

The inner joins (theta join, equijoin, and natural join) include only those tuples with matching attributes and the rest are discarded in the resulting relation.

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples **that do not match** the criteria.

There are three kinds of **outer joins** −

- left outer join (⟕)
- right outer join (⟖)
- full outer join (⟗)

LEFT OUTER JOIN (A ⟕ B)

The left outer join operation allows **keeping all tuples in the left relation**. However, if there is no matching tuple is found in **right relation**, then the attributes of right relation in the join result are **filled with null values**.

Course

Course_ID	Course_Name
101	Database
102	Networking
103	AI

Faculty

Course_ID	Faculty_Name
101	A
103	B
105	C

Course ⟕ Faculty

Course_ID	Course_Name	Faculty_Name
101	Database	A
102	Networking	NULL
103	AI	B

RIGHT OUTER JOIN: (A ⟖ B)

In the right outer join, operation allows **keeping all tuple in the right relation**. However, if there is no matching tuple is found in the **left relation**, then the attributes of the left relation in the join result are **filled with null values.**

Course

Course_ID	Course_Name
101	Database
102	Networking
103	AI

Faculty

Course_ID	Faculty_Name
101	A
103	B
105	C

Course ⋈ Faculty

Course_ID	Faculty_Name	Course_Name
101	A	Database
103	B	AI
105	C	NULL

FULL OUTER JOIN: (A ⋈ B)

Course

Course_ID	Course_Name
101	Database
102	Networking
103	AI

Faculty

Course_ID	Faculty_Name
101	A
103	B
105	C

Course ⋈ Faculty

Course_ID	Course_Name	Faculty_Name
101	Database	A
102	Networking	NULL
103	AI	B
105	NULL	C

Chapter-5

RELATIONAL DATABASE DESIGN

DATABASE DESIGN AND NORMAL FORMS

In database design coming up with a **'good' schema** is very important.

- How do we characterize the "goodness" of a schema?
- If two or more alternative schemas are available, how do we compare them?
- What are the problems with **'bad'** schema designs?

Example-

Consider the data about employees and departments is to be stored,

Details of employee are: EmpName, EmpID, Gender, EmpDept

Details of department are: DeptName, Manager, Location

Several employees belong to a department.

EmpDept (Bad schema)

EmpID	EmpName	Gender	DeptName	Manager	Location
101	Sachin	M	Software	Saurabh	Bldg-A
102	Virat	M	Software	Saurabh	Bldg-A
204	Rohit	M	HR	Radha	Bldg-B

RELATIONAL DATABASE DESIGN

The **goal of relational database design** is to generate a **set of relation schemas** that allows to store information without **unnecessary redundancy**, yet also allows to **retrieve information easily**.

This is accomplished by **designing schemas** that are in an appropriate **normal form**.

Normal Forms:

- Each normal form specifies certain conditions.
- If the conditions are satisfied by the schema, certain kind of problems are avoided.
- A **formal approach to relational database design** is based on the notion of **functional dependencies**.
- Normal forms are then defined in terms of **functional dependencies and other types of data dependencies.**

DECOMPOSITION

A decomposition is the process of **breaking down the relations** into progressively greater (finer and finer) levels of detail.

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

Decomposition helps in **eliminating some of the problems of bad design** such as redundancy, inconsistencies and anomalies.

There are **two types of decomposition:**

- **Lossy Decomposition**
- **Lossless Join Decomposition**

Example-Decomposition

'Good' schema is an example of **decomposition.**

Employee

EmpID	EmpName	Gender	EmpDept
101	Sachin	M	Software
102	Virat	M	Software
204	Rohit	M	HR

Department

DeptName	Manager	Location
Software	Saurabh	Bldg-A
HR	Radha	Bldg-B

Decomposition is the only way to avoid the redundancy in the **EmpDept** schema.

It decomposes the relation **EmpDept** into two schemas- **Employee** and **Department** schemas.

Lossy Decomposition:

The decomposition of relation R into R1 and R2 is **lossy** when the **join of R1 and R2** does not **yield the same relation** as in R.

In lossy decomposition, **some information is lost during retrieval** of original relation or table as we cannot **reconstruct** the original relation.

Consider a **Student** relation as

StdID	Name	Department
101	Aniket	Computer
119	Aniket	Mechanical

We can decompose **Student** relation as

Student1

StdID	Name
101	Aniket
119	Aniket

Student2

Name	Department
Aniket	Computer
Aniket	Mechanical

The problem arises when we have two students with the same name.

In **lossy** decomposition, spurious tuples are generated when a **natural join** is applied to the relations in the decomposition.

For example: consider the natural join of Student1 and Student2 relations as

Student1 **natural join** Student2

Result

StdID	Name	Department
101	Aniket	Computer
101	Aniket	Mechanical
119	Aniket	Computer
119	Aniket	Mechanical

The above decomposition is a **bad decomposition or Lossy decomposition**.

Lossless Join Decomposition

The decomposition of relation **R into R1 and R2** is **lossless** when the join of R1 and R2 yield the same relation as in R.

A relational table is **decomposed into two or more smaller tables**, in such a way that the contents of the original table can be **reconstructed** by joining the decomposed parts. This is called **lossless-join** decomposition as there is **no loss of information**.

Suppose **Student** relation is decomposed into two relations- Stud_name and Stud_dept

Student

StdID	Name	Department
101	Aniket	Computer
119	Aniket	Mechanical

Std_name

StdID	Name
101	Aniket
119	Aniket

Std_dept

StdID	Department
101	Computer
119	Mechanical

When these two relations are **joined** on the common column **StdID** as

Std_name **natural join** Std_dept

result will be

StdID	Name	Department
101	Aniket	Computer
119	Aniket	Mechanical

In lossless decomposition, no any spurious tuples are generated when a natural joined is applied to the relations in the decomposition.

ANOMALIES - PROBLEMS WITH BAD SCHEMA

There are **three types of anomalies** that occur when the database is not normalized. These are –

Insertion, update and deletion anomaly.

EmpDept

EmpID	EmpName	DeptName	Manager	Location
101	Sachin	Software	Saurabh	Bldg-A
102	Virat	Software	Saurabh	Bldg-A
204	Rohit	HR	Radha	Bldg-B

Insertion anomaly:

No way of inserting info about a new department unless we also enter details of an employee in department.

Deletion anomaly:

If all employees of a certain department leave and we delete their tuples, information about the department itself is lost.

Update Anomaly:

Updating information about manager of department

- value in several tuples needs to be changed and
- if a tuple is missed, data will be in inconsistency state.

NORMAL FORMS

Normalization is a method to overcome all these anomalies and bring the database to a consistent state.

The different forms of normalization are:

First Normal Form (1NF) - included in the definition of a relation

Second Normal Form (2NF) - defined in terms of functional dependencies

Third Normal Form (3NF) - defined in terms of functional dependencies

Fourth Normal Form (4NF) - defined using multi-valued dependencies

Fifth Normal Form (5NF) or Project Join Normal Form (PJNF) - defined using join dependencies

NORMALIZATION THEORY

Normalization decides whether a particular **relation R** is in **"good" form**.

In case a relation **R** is **not in "good"** form, **decompose it** into set of relations

{R_1, R_2, ..., R_n} such that

- each relation is in **good form** and
- decomposition is a **loss-less decomposition**.

Normalization theory is based on:

- **Functional dependencies**
- **Multivalued dependencies**

Functional Dependency plays a vital role to find the difference between **good** and **bad** database design.

FUNCTIONAL DEPENDENCIES

There are usually a variety of **constraints (rules) on the data** in the real world.

For example, some of the **constraints** that are expected to hold in a **university database** are:

- Students and faculty are uniquely identified by their ID.
- Each student and faculty has only one name.
- Each faculty and student is (primarily) associated with only one department.
- Each department has only one value for its budget, and only one associated building.

Legal instance of relation and database

An **instance of a relation** that satisfies all such **real-world constraints** is called a **legal instance** of the relation.

A **legal instance of a database** is one where all the relation instances are **legal instances.**

Constraints on the **set of legal relations** require that the value for a **certain set of attributes determines** uniquely the **value for another set of attributes**.

A **functional dependency** is a generalization of the notion of a **key (primary key).**

Definition - Functional Dependencies

Functional Dependency (FD) determines the relationship of one attribute to another attribute.

Let R be a relation schema **$X \subseteq R$ and** $Y \subseteq R$

The **functional dependency (FD)**

$X \rightarrow Y$ **(**read as : **X functionally determines Y)**

holds (true) on R if and only if for any legal relations r(R), whenever any two tuples **t_1 and t_2 of r** agree on the attributes X, they also agree on the attributes Y.

That is, **$t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$**

$X \subseteq R$ indicates **X is subset of R** that is every element of X is in R.

$Y \subseteq R$ indicates **Y is subset of R** that is every element of Y is in R.

Functional Dependencies-Examples

If **$K \subset R$** (K is proper subset of R) is a **primary key for R**, then

for **any $A \in R$** (A is element of R, but R has more elements),

$K \rightarrow A$ (K determines A) holds **true**.

Consider the schema:

Student (Stud_name, <u>Roll_no</u>, Dept_name, Hostel_name, Room_no)

Since **Roll_no is a primary key**,

Roll_no → {Stud_name, Dept_name, Hostel_name, Room_no}

Suppose that each student is given a hostel room exclusively, then

Hostel_name, Room_no → Roll_no

Consider the schema:

inst_dept (ID, name, salary, dept_name, building, budget)

in which the functional dependency (FD)

dept_name → budget

holds (true) because for **each department** (identified by dept_name), there is a **unique budget** amount.

We denote the fact that the pair of attributes (ID, dept name) forms a superkey for inst_dept by writing:

ID, dept_name → {name, salary, building, budget}

FDs are additional constraints that can be specified by database designers.

Functional dependencies allow us to express constraints that cannot be expressed using **superkeys**.

Consider the schema:

EmpDept

EmpID	EmpName	Salary	DeptName	Location	Budget
101	Sachin	65000	Software	Bldg-A	50 Lacs
109	Virat	95000	Software	Bldg-A	30 Lacs
204	Rohit	80000	HR	Bldg-B	35 Lacs

We expect these functional dependencies to hold true:

DeptName→ Location

EmpID → Location

but would not expect the following to hold true:

DeptName → Salary

CLOSURE OF A SET OF FUNCTIONAL DEPENDENCIES

Given a set **F set of functional dependencies**, there are certain other functional dependencies that are logically implied by F.

If A → B and B → C, then we can infer that A → C.

The set of **all** functional dependencies logically implied by F is the **closure** of F.

The closure of F is denoted by $\mathbf{F^+}$ and it contains all of the functional dependencies in F.

Consider an example,

Employee

EmpID	EmpName	Gender	EmpDept
101	Sachin	M	Software
102	Virat	M	Software
204	Rohit	M	HR

closure of F (F^+) for Employee schema

EmpId → EmpName

EmpName → EmpDept

EmpId → EmpDept

TYPES OF FUNCTIONAL DEPENDENCIES

- Multivalued dependency
- Trivial functional dependency
- Non-trivial functional dependency
- Transitive dependency

Multivalued dependency

Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table.

Example:

Car_model	Manf_year	Color
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue

In this example, manf_year and color are independent of each other but dependent on Car_model. These two columns are said to be multivalued dependent on Car_model.

This dependence can be represented like this:

Car_model $\rightarrow$ Manf_year

Car_model $\rightarrow$ Color

Trivial Functional Dependency

If a functional dependency (FD) $X \rightarrow Y$ holds, where Y is a subset of X ($Y \subseteq X$), then it is called a **trivial FD**. Trivial FDs always hold true.

The dependency of an attribute on a set of attributes is known as trivial functional dependency **if the set of attributes includes that attribute**.

Example: For Employee relation,

Emp_ID	Emp_name
101	Sachin
218	Rahul
319	Rohit

the dependency **{Emp_ ID, Emp_name} → Emp_ ID**

is a trivial functional dependency as Emp_ID is a subset of {Emp_ ID,Emp_name}.

That means if we know the values of Emp_ID and Emp_name then the value of Emp_ID can be **uniquely determined**.

Following are also trivial dependencies,

Student_Id → Student_Id and Student_Name → Student_Name.

Non-trivial functional dependency

If an FD X → Y holds, where Y is not a subset of X (Y ⊄X) ,then it is called a non-trivial FD.

For example: relation **Employee**

Emp_ID	**Emp_name**	**Emp_address**
101	Sachin	Mumbai
218	Rahul	Navi Mumbai
319	Rohit	Thane

The following functional dependencies are **non-trivial**:

Emp_ID → Emp_name

where Emp_name is not a subset of Emp_ID &

Emp_ID → Emp_address

where Emp_address is not a subset of Emp_ID

Completely non-trivial functional dependency

If an FD X → Y holds, where x intersect Y(X ∩Y) = Φ,

it is said to be a completely non-trivial FD.

Transitive dependency

A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies. A transitive dependency can only occur in a relation of three of more attributes.

Definition: An FD **X →Y** in a relation schema R for which there is a set of attributes Z ⊆R such that **X →Z** and **Z →Y** and Z is not a subset of any key of R.

That is, if **X →Z** and **Z →Y** in a relation schema R, then **X →Y.**

Example:

Book	**Author**	**Author_age**
Wise & Otherwise	Sudha Murthy	70
Harry Potter	J. K. Rowling	55
Room on the roof	Ruskin Bond	86

Thus, if **{Book} → {Author}** and **{Author} → {Author_age}**

Then **{Book} → {Author_age}**

In Transitive Dependency, a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

INFERENCE RULES (IR) OR ARMSTRONG'S AXIOMS:

- The Armstrong's axioms are the basic inference rules.
- Armstrong's axioms are used to conclude functional dependencies on a relational database.
- It can apply to a set of FD (functional dependency) to derive other FD.
- The closure of F (F^+) contains all of the functional dependencies in F (set of FDs).

The Functional dependency has 3 types of inference rules:

1. **Reflexive Rule (IR_1)**

   If X is a set of attributes and $Y \subseteq X$ (Y is subset of X), then $X \rightarrow Y$ holds true.

2. **Augmentation Rule (IR_2)**

   If $X \rightarrow Y$ holds and Z is a set of attributes, then $XZ \rightarrow YZ$ holds

3. **Transitive Rule (IR_3)**

   If $X \rightarrow Y$ holds and $Y \rightarrow Z$ holds, then $X \rightarrow Z$ holds.

Additional 3 inference rules of functional dependency which can be derived from the first 3 axioms are :

4. **Union Rule (IR_4)**

   If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$.

5. **Decomposition Rule (IR_5)**

   If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

6. **Pseudo transitive Rule (IR_6)**

   If $X \rightarrow Y$ and $YZ \rightarrow W$ then $XZ \rightarrow W$

Armstrong's axioms are said to be sound and complete as

- they are sound, because they do not generate any incorrect functional dependencies.
- they are complete, because, for a given set F of functional dependencies, they allow us to generate all F+.

NORMALIZATION

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations.
- It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The inventor of the relational model E. Codd proposed the theory of normalization with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form.
- Later he joined R. F. Boyce to develop the theory of Boyce-Codd Normal Form.

The Theory of Normalization in SQL is still being developed further. However, in most practical applications**, normalization achieves its best in 3rd Normal Form**.

FIRST NORMAL FORM (1NF)

Each column of relation should be **single valued (indivisible values)** which means it should not contain **multiple values**. Such attribute domains are said to have **atomic values**.

Example: Consider a relation **Student** with attributes student's roll no, name and subjects opted.

Roll_No	**Name**	**Subject**
101	Rahul	OS, CN
103	Sachin	Java
102	Virat	C, C++

The relation **Student** in **First Normal Form (1NF) can be**

Roll_No	**Name**	**Subject**
101	Rahul	OS
101	Rahul	CN
103	Sachin	Java
102	Virat	C
102	Virat	C++

SECOND NORMAL FORM (2NF)

For a relation to be in the Second Normal Form,

1. it should be in the **First Normal form** and
2. it should **not have partial dependency** or every **non-prime attribute** is **fully functionally dependent on primary key**.

Full functional dependency:

- An FD **X →A** for which there is **no proper subset** Y of X such that **Y →A**

i.e. A is said to be fully functionally dependent on X.

Partial Dependency :

- An FD **X →A** for which **there is subset** Y of X such that **Y →A**.

Prime attribute : A attribute that is key or part of some key (primary).

Example: ERP_ID in Student relation or {Emp_ID, Dept_ID} in Employee relation

Non-prime attribute: An attribute that is not part of any key

Example: Marks, Address in Student relation or Dept_location,Salary in Employee relation

Consider an example of partial dependency

Score

Student_ID	Subject_ID	Subject_Name	Marks
101	CSC01	C++	70
102	CSC02	DBMS	75
102	CSC03	Java	80

Since **Primary key= {Student_ID, Subject_ID},** following FD is true

{Student_ID, Subject_ID} → Marks

But this FD is not true,

{Student_ID, Subject_ID} → Subject_Name

This FD is not true, because non-prime attribute (**Subject_Name)** is not functionally dependent on primary key.

For Subject_Name, following FD is true

Subject_ID → Subject_Name

This is called as **partial dependency** where non-prime attributes are functionally dependent on part of the primary key and not whole key.

Thus the solution is to decompose the Score relation into two relations Score and Subject as

Score

Student_ID	Subject_ID	Marks
101	CSC01	70
102	CSC02	75
102	CSC03	80

Following FD is true,

{Student_ID, Subject_ID} → Marks

Subject

Subject_ID	Subject_Name
CSC01	C++
CSC02	DBMS
CSC03	Java

Following FD is true,

Subject_ID → Subject_Name

Score and Subject relations are now in the **Second Normal Form**, with **no partial dependency** where non-prime attributes are functionally dependent on whole primary key.

THIRD NORMAL FORM (3NF)

A relation is said to be in the Third Normal Form when,

1. it is in the Second Normal form and
2. it **doesn't have Transitive Dependency**.

Example: For Book relation,

Book	Author	Author_age
Wise & Otherwise	Sudha Murthy	70
Harry Potter	J. K. Rowling	55
Room on the roof	Ruskin Bond	86

transitive dependency can be represented as

{Book} → {Author} and **{Author} → {Author_age}**

Hence **{Book} → {Author_age}.**

In **Transitive Dependency,** a **non-prime attribute** depends on **other non-prime attributes** rather than depending upon the **prime attributes or primary key.**

Thus the solution is to decompose the Book relation into two relations Book and Author as

Book

Book	**Author**
Wise & Otherwise	Sudha Murthy
Harry Potter	J. K. Rowling
Room on the roof	Ruskin Bond

Author

Author	**Author_age**
Sudha Murthy	70
J. K. Rowling	55
Ruskin Bond	86

Following FDs are true,

Book → Author

Author → Author_age

Book and Author relations are in **Third Normal Form**, with **no Transitive dependency.**

BOYCE AND CODD NORMAL FORM (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form.

A relation schema R is in **BCNF** with respect to a set **F of functional dependencies** if, for all functional dependencies in F+ of the form X → Y, where X ⊆ R and Y ⊆ R, **at least one of the following holds true:**

- **X → Y** is a **trivial functional dependency** (that is, Y ⊆ X).
- **X is a superkey for schema R**.

BCNF does not allow **non-prime attribute** to determine **prime attributes**.

Example: Consider the relation **EmpDept as**

EmpID	EmpName	Salary	DeptName	Location	Budget
101	Sachin	65000	Software	Bldg-A	50 Lacs
109	Virat	95000	Software	Bldg-A	30 Lacs
204	Rohit	80000	HR	Bldg-B	35 Lacs

The functional dependency (FD) **dept_name → budget** holds (true) because for **each department** (identified by dept_name), there is a **unique budget** amount.

But **dept_name** is not a superkey (because, a department may have a number of different employees and dept_name will be mentioned in those many tuples).

The decomposition of **EmpDept** into **Employee** and **Department** is a better design.

Employee

EmpID	EmpName	Salary	DeptName
101	Sachin	65000	Software
109	Virat	95000	Software
204	Rohit	80000	HR

Department

DeptName	Location	Budget
Software	Bldg-A	50 Lacs
Software	Bldg-A	30 Lacs
HR	Bldg-B	35 Lacs

The **Employee** and **Department** schema is in BCNF.

FOURTH NORMAL FORM (4NF)

A relation is said to be in the **Fourth Normal Form** when,

1. It is in the **Boyce-Codd Normal Form**.
2. It **doesn't** have **Multi-valued Dependency**.

Example: Consider a student ENROLMENT relation

Std_ID	**Course**	**Hobby**
101	Sociology	Cricket
101	History	Hockey
102	Economics	Cricket
103	Statistics	Hockey

Student with Std_ID -101 has opted for two courses, Sociology and History, and has two hobbies, Cricket and Hockey.

101	Sociology	Cricket
101	History	Hockey

There is no relationship between the columns, Course and Hobby. They are independent of each other.

So there is multi-valued dependency, which leads to un-necessary repetition of data.

To make the above relation to satisfy the Fourth Normal Form, decompose it into 2 tables as:

StdCourses

Std_ID	**Course**
101	Sociology
101	History
102	Economics
103	Statistics

StdHobbies

Std_ID	**Hobby**
101	Cricket
101	Hockey
102	Cricket
103	Hockey

Thus, StdCourses and StdHobbies relations are in Fourth Normal Form.

Chapter-6

Transactions Management, Concurrency and Recovery

INTRODUCTION

Collections of operations that form a **single logical unit of work** are called **transactions**.

A database system must ensure **proper execution of transactions** despite failures -

either the entire transaction executes, or none of it does.

Furthermore, it must manage concurrent execution of transactions in a way that avoids inconsistency of results.

Example:

a transfer of funds from one account to another account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations.

Transaction Definition

A **transaction** is a **unit** of **program execution** that accesses and possibly updates various data items.

Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in JDBC or ODBC.

A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**.

The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

PROPERTIES OF TRANSACTIONS (ACID PROPERTIES)

1. A (ATOMICITY)

Either all operations of the transaction are reflected properly in the database, **or none are**.

As a collection of steps, transaction must appear to the user **as a single, indivisible unit**.

Since a transaction is indivisible, it **either executes in its entirety or not at all**.

Thus, if a transaction begins to execute but fails for whatever reason, **any changes** to the database that the transaction may have made **must be undone**.

This requirement holds **regardless of whether**

- the transaction itself failed (for example, if it divided by zero),
- the operating system crashed, or
- the computer itself stopped operating.

2. C (CONSISTENCY)

Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the **consistency of the database**.

A transaction must preserve database **consistency** - if a transaction is run atomically in isolation starting from a consistent database, the **database must again be consistent** at the end of the transaction.

Maintaining consistency in transaction execution is the **responsibility of the programmer** who codes a transaction.

3. I (ISOLATION)

Since a transaction is a single unit, **its actions cannot appear to be separated by other** database operations not part of the transaction.

Even though **multiple transactions** may execute **concurrently**, the system guarantees that, for every pair of transactions T1 and T2, it appears to T1 that either T2 finished execution before T1 started or T2 started execution after T1 finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Even a single SQL statement involves many separate accesses **(read/write)** to the database, and a transaction may consist of **several SQL statements**.

Therefore, the database system must take special actions to ensure that **transactions operate properly without interference from other concurrently executing database statements**.

4. D (DURABILITY)

After a transaction completes **successfully**, the **changes** it has made to the database **persist**, even if there are system failures.

Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system **"forgets"** about the transaction.

Thus, a transaction's actions **must persist** across crashes. This property is referred to as **durability**.

TRANSACTION MANAGEMENT

Transaction management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

Concurrency control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Transaction Model

Consider a simple bank application consisting of several accounts and a set of transactions that access and update those accounts.

Transactions access data using **two operations**:

- **Read (X) - read** operation

it transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the **read** operation.

- **Write (X) - write** operation

it transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

Assume that the **write** operation updates the database immediately.

Example: transaction

Consider an account A with ₹ 950 as balance while account B with ₹ 1050 as balance.

Let T1 be a transaction that transfers ₹ 50 from account A to account B.

The transaction T1 can be defined as:

```
T1: read(A);
   A := A − 50;
   write(A);
   read(B);
   B := B + 50;
   write(B).
```

ACID PROPERTIES

Consistency:

As a consistency requirement, the **sum of A and B before an execution** of the transaction and **after the execution** of the transaction **should be unchanged**.

For money transfer example, sum of balance in accounts A and B,

- before an execution of the transaction (₹ 950 + ₹ 1050 = ₹ 2000) and
- after the execution of the transaction (₹ 900 + ₹ 1100 = ₹ 2000)

should be unchanged.

Ensuring consistency for an individual transaction is the responsibility of the **application programmer** who codes the transaction.

This task may be facilitated by **automatic testing of integrity constraints**.

Atomicity:

Suppose that, during the execution of transaction T1, a **failure occurs** that prevents T1 from completing its execution successfully. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation.

In this case, the values of accounts A and B reflected in the database are ₹900 and ₹1050 (instead of ₹900 and ₹1100). The system **destroyed ₹50** as a result of this failure.

In particular, we note that the sum A + B is **no longer preserved**.

Thus, because of the failure, the state of the system no longer reflects a **real state** of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such **inconsistencies are not visible** in a database system.

If the **atomicity** property is present, **all actions** of the transaction are reflected in the database, **or none** are.

Ensuring atomicity is **handled by** a component of the database called the **recovery system.**

Durability:

Once the execution of the transaction **completes successfully**, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that **no system failure can result in a loss of data** corresponding to this transfer of funds.

The **durability** property guarantees that, once a **transaction completes successfully**, all the **updates** that it carried out on the database **persist, even if there is a system failure after the transaction completes execution**.

We can guarantee durability by ensuring that either:

- The updates carried out by the transaction have been written to disk before the transaction completes.
- Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

The **recovery system** of the database is responsible for ensuring durability.

Isolation:

If several transactions are executed concurrently, **their operations may interleave** in some **undesirable way**, resulting in an **inconsistent state**.

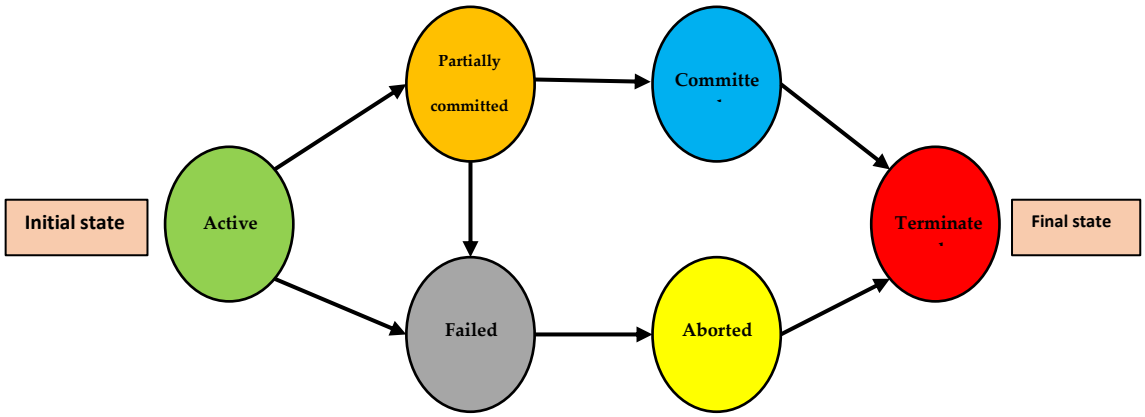
For **example**, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate

point and computes A+B, it will observe an **inconsistent value**. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

The **isolation** property of a transaction ensures that the **concurrent execution** of transactions results in a system state that **is equivalent to** a state that could have been obtained had these transactions **executed one at a time in some order**.

Ensuring the isolation property is the responsibility of a **concurrency-control system.**

TRANSACTION STATE DIAGRAM



A transaction **starts in the active state**.

When it finishes its final statement, it enters the partially **committed state**.

At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state.

At this point, the system has two options: **restart** the transaction **or kill** the transaction.

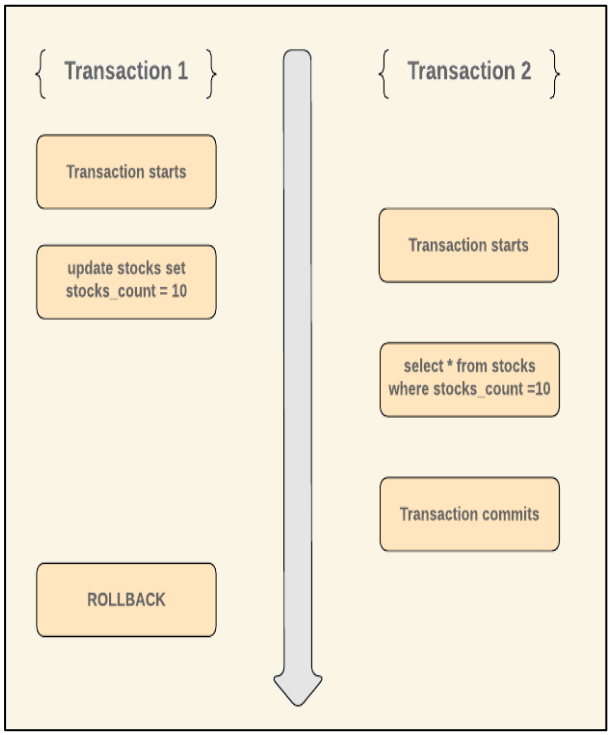
CONCURRENT EXECUTIONS

Transaction processing systems usually allow **multiple transactions to run concurrently**.

There are **two reasons** for allowing concurrency:

- **Improved throughput and resource utilization:**
    - A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. This increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time.
    - Correspondingly, the processor and disk **utilization** also increase; that is , the processor and disk spend less time idle, or not performing any useful work.
- **Reduced waiting time** for transactions: short transactions need not wait behind long ones. Moreover, it also reduces the **average response time**, that is , the average time for a transaction to be completed after it has been submitted.

Example- Concurrent Executions



SCHEDULES

Schedule is a **sequence of instructions** that specify the **chronological order** in which instructions of **concurrent transactions are executed**.

- A schedule for a set of transactions must **consist of all instructions** of those transactions
- Must **preserve the order** in which the instructions appear in each individual transaction.
- A transaction that **successfully completes** its execution will have a **commit** instruction as the last statement.
- A transaction **that fails to successfully complete** its execution will have an **abort** instruction as the last statement.

Schedule can be either

- **Serial**
- **Concurrent**

Example: Serial Schedule

Let T1 and T2 be two transactions that transfer funds from one account to another.

It is defined as:

```
T1: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).
```

Transaction T2 transfers 10 percent of the balance from account A to account B.

It is defined as:

```
T2: read(A);
temp := A * 0.1;
A := A − temp;
write(A);
read(B);
B := B + temp;
write(B).
```

Schedule-1

Suppose the current values of accounts A and B are ₹1000 and ₹2000, respectively.

Suppose also that the two transactions are executed one at a time in the order T1 followed by T2. This execution sequence appears as in schedule-1,

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

The final values of accounts A and B, after the execution T1 and T2 takes place, are ₹855 and ₹2145, respectively. Thus, the total amount of money in accounts A and B, that is, the sum A + B is preserved after the execution of both transactions.

Schedule 1- a serial schedule in which T1 is followed by T2.

Schedule-2

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is as in schedule-2,

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Again, the sum A + B is preserved, and the final values of accounts A and B are ₹850 and ₹2150, respectively.

Schedule 2- a serial schedule in which T2 is followed by T1.

These schedules (Schedule-1 & Schedule-2) are **serial schedules:**

Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

CONCURRENT SCHEDULE

- When the database system executes **several transactions** concurrently, the **corresponding schedule no longer needs to be serial**.
- If two transactions are running concurrently, the operating system may execute **one transaction for a little while**, then perform a context switch, execute the **second transaction for some time**, and then switch back to the first transaction for some time, and so on.
- With multiple transactions, the **CPU time is shared among all the transactions**.

The **concurrency-control** component of the database system ensures that any schedule that is executed will leave the database in a consistent state.

Schedule-3: Concurrent Schedule

Suppose that the two transactions T1 and T2 are executed concurrently.

One possible schedule-3 is

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Schedule 3 is not a serial schedule.

After this execution takes place, it will give same state as the one in which the transactions are executed serially in the order T1 followed by T2 (schedule-1).

The sum A + B is indeed preserved.

Schedule 3- a concurrent schedule equivalent to schedule 1.

Schedule-4: Concurrent Schedule with inconsistent state

In Schedules 1, 2 and 3, the **sum A + B is preserved**. But not all concurrent executions result in a correct (consistent) state.

Consider a schedule-4 as

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

After the execution of this schedule, we arrive at a state where the final values of accounts A and B are ₹950 and ₹2100, respectively.

This final state is an **inconsistent state**, since we have gained ₹50 in the process of the concurrent execution.

The sum A + B is not preserved by the execution of the two transactions.

Schedule 4-a concurrent schedule resulting in an inconsistent state.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state,

such as the one just described, are possible.

The **concurrency-control** component of the database system carries out this task.

SERIALIZABILITY

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

Different forms of schedule equivalence give rise to the notions of:

1. **Conflict serializability**
2. **View serializability**

Simplified view of transactions

We ignore operations other than **read** and **write** instructions.

We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

Our simplified schedules consist of only **read** and **write** instructions.

We ignore operations other than **read** and **write** instructions.

We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Schedule-3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

can be represented as

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 6 - a serial schedule that is equivalent to schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

CONFLICTING INSTRUCTIONS

Consider a schedule S in which there are two **consecutive instructions**, instructions I_1 and I_2, of transactions T_1 and T_2 respectively ($I_1 \neq I_2$). If I_1 and I_2 refer to **different data items**, then **we can swap I_1 and I_2 without affecting the results** of any instruction in the schedule. However, if I_1 and I_2 refer to the **same data item Q**, then the **order of the two steps may matter**.

With only read and write instructions, **there are four cases that we need to consider**:

T_1	T_2	Conflict or not?
I_1 = read(Q)	I_2 = read(Q)	I_1 and I_2 don't conflict
I_1 = read(Q)	I_2 = write(Q)	I_1 and I_2 conflict
I_1 = write(Q)	I_2 = read(Q)	I_1 and I_2 conflict
I_1 = write(Q)	I_2 = write(Q)	I_1 and I_2 conflict

Thus, only in the case where both I_1 and I_2 are **read instructions,** the **order of their execution does not matter**.

Thus, I_1 and I_2 **conflict** if they are instructions in different transactions **on the same data item**, and **at least one** of these instructions **is a write operation**.

In **Schedule-3**

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

The write(A) instruction of T1 **conflicts** with the read(A) instruction of T2 as they are instructions in **different transactions on the same data item**, and **one** of these instructions **is a write operation.**

However, the write(A) instruction of T2 **does not conflict** with the read(B) instruction of T1, because the two instructions access different data items.

NONCONFLICTING INSTRUCTIONS

Let $\mathbf{I_1}$ and $\mathbf{I_2}$ be consecutive instructions of a schedule S.

If $\mathbf{I_1}$ and $\mathbf{I_2}$ are instructions of different transactions and $\mathbf{I_1}$ and $\mathbf{I_2}$ do not conflict, then we can swap the order of $\mathbf{I_1}$ and $\mathbf{I_2}$ to produce a new schedule S'.

S is equivalent to S', since all instructions appear in the same order in both schedules except for $\mathbf{I_1}$ and $\mathbf{I_2}$, whose order does not matter.

In **Schedule-3**

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Since the write(A) instruction of T2 in schedule-3 **does not conflict** with the read(B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule-5.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule-3 is equivalent schedule-5 and both produce the same final system state.

We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T1 with the read(A) instruction of T2.
- Swap the write(B) instruction of T1 with the write(A) instruction of T2.

- Swap the write(B) instruction of T1 with the read(A) instruction of T2.

The final result of these swaps, schedule 6 is a serial schedule (same as schedule-1)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

CONFLICT EQUIVALENT SCHEDULES

If a schedule **S** can be **transformed** into a schedule **S'** by a series of **swaps of non-conflicting instructions**, we say that S and S' are **conflict equivalent**.

Schedule 3 and Schedule 5 are **conflict equivalent.**

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5

CONFLICT SERIALIZABLE SCHEDULE

The concept of conflict equivalence leads to the concept of conflict serializability.

We say that a schedule S is **conflict serializable** if it is conflict equivalent to a **serial schedule**.

Thus, schedule-3 is **conflict serializable**, since it is **conflict equivalent** to serial schedule-1.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

schedule-3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule-1

VIEW SERIALIZABLE SCHEDULE

The concept of view equivalence leads to the concept of view serializability.

We say that a schedule S is **view serializable** if it is view equivalent to a serial schedule.

Every conflict serializable schedule is also view serializable.

Precedence graph

Precedence graph is simple and efficient method for determining conflict serializability of a schedule.

Consider a schedule **S**.

We construct a **directed graph**, called a **precedence graph**, from schedule S.

This graph consists of a

pair G = (V, E),

where

V is a set of vertices and

E is a set of edges.

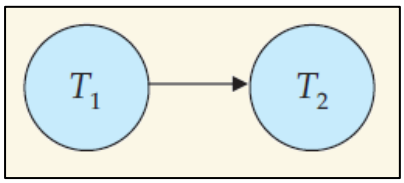
The set of vertices consists of all the transactions participating in the schedule.

Example - precedence graph

schedule-1 is as shown below,

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

The precedence graph for schedule-1 is,

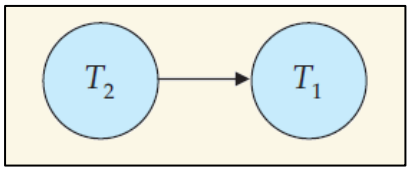


Above precedence graph contains the single edge **T1 → T2**, since all the instructions of T1 are executed before the first instruction of T2 is executed.

schedule-2 is as shown below

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	

The precedence graph for schedule-2 is

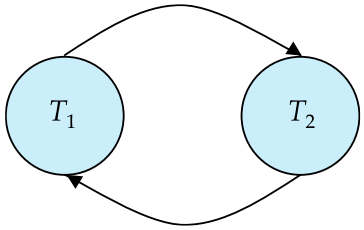


Above precedence graph shows the precedence graph for schedule 2 with the single edge **T2 → T1**, since all the instructions of T2 are executed before the first instruction of T1 is executed.

Schedule-4 is as shown below

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

The precedence graph for schedule-4 is



The precedence graph contains the edge **T1 →T2**, because T1 executes read(A) before T2 executes write(A). It also contains the edge **T2→T1**, because T2 executes read(B) before T1 executes write(B).

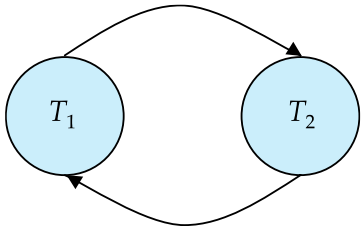
TESTING FOR SERIALIZABILITY

If the **precedence graph** for S has a **cycle**, then schedule S is **not conflict serializable**.

If the graph contains **no cycles**, then the schedule S is **conflict serializable**.

For example, the precedence graph for schedule-4 as shown below and **has a cycle**,

then schedule-4 is said to be **not conflict serializable** schedule.



LOCK-BASED PROTOCOLS

One way to ensure isolation is to require that data items be accessed in a **mutually exclusive manner**; that is, while **one transaction is accessing a data item, no other transaction can modify that data item**.

The most common method used to implement this requirement is to allow a transaction to access a data item **only if it is currently holding a lock on that item**.

Locks

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

Data items can be locked in **two modes:**

1. **Shared mode-** (denoted by **S**)

   If a transaction T1 has obtained a **shared-mode lock** on item Q, then T1 **can read**, but **cannot write**, Q.

   Shared lock is requested using **lock-S** instruction.

2. **Exclusive** - (denoted by **X**)

   If a transaction T1 has obtained an **exclusive-mode lock** on item Q, then T1 **can both read and write** Q.

   Exclusive lock is requested using **lock-X** instruction.

Command **unlock(X)** is used to free data item **X** from locking.

The use of these two lock modes allows **multiple transactions to read a data item** but limits **write access to just one transaction at a time**.

Every transaction **request** a **lock** in an **appropriate mode** on data item Q, depending on the types of operations that it will perform on Q.

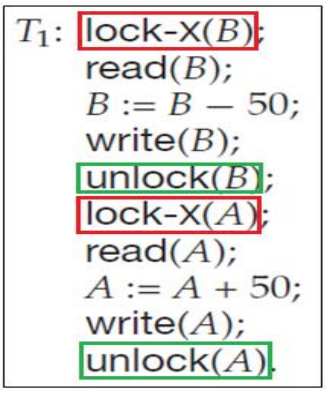
The transaction makes the request to the **concurrency-control manager**.

The transaction can proceed with the operation only after the concurrency-control manager **grants the lock** to the transaction.

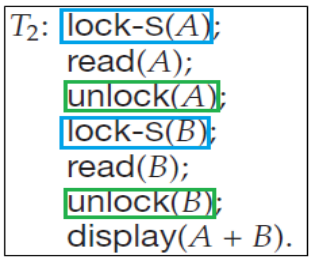
Banking Example- Lock-Based Protocols

Let A and B be two accounts that are accessed by transactions T1 and T2.

Transaction T1 transfers ₹50 from account B to account A.



Transaction T2 displays the total amount of money in accounts A and B that is, sum (A + B).



Suppose that the values of accounts A and B are ₹100 and ₹200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value ₹300.

CONCURRENCY CONTROL

Concurrency Control schemes to ensure isolation of transactions are:

- **Lock-based protocols,**
- **Timestamp-based protocols**

TWO-PHASE LOCKING PROTOCOL

Two-Phase Locking Protocol ensures **conflict-serializable** schedules.

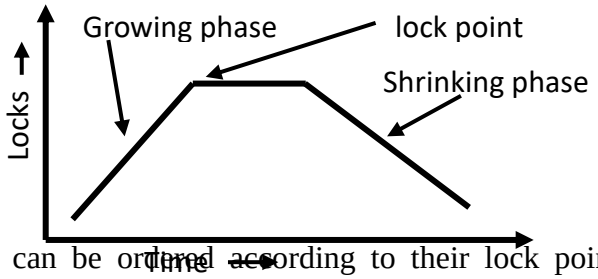
In this protocol, each transaction issues **lock and unlock requests in two phases**:

1. Growing phase - A transaction **may obtain locks**, but may not release any lock.

2. Shrinking phase - A transaction **may release locks**, but may not obtain anynew locks.

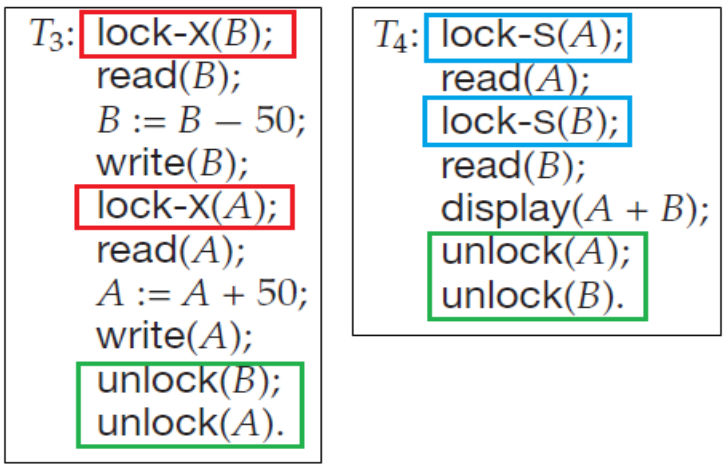
- **Initially**, a transaction is in the **growing phase**.
- The transaction **acquires locks** as needed.
- Once the transaction **releases a lock**, it enters the **shrinking phase**, and it can issue **no more lock requests**.

In any transaction, **the point in the schedule** where the transaction has obtained its final lock (the **end of its growing phase**) is called the **lock point** of the transaction.



The transactions can be ordered according to their lock points - this ordering is, in fact, a **serializability ordering** for the transactions.

For example,



Transactions T3 and T4 are **two phase** because they have followed growing and shrinking phase locking and unlocking.

Types of Two-Phase Locking Protocol

- **Strict two-phase locking:**

  A transaction must **hold all its exclusive locks till it commits/aborts**.

  Ensures recoverability and avoids cascading roll-backs.

- **Rigorous two-phase locking**:

  A transaction must **hold all locks till commit/abort**.

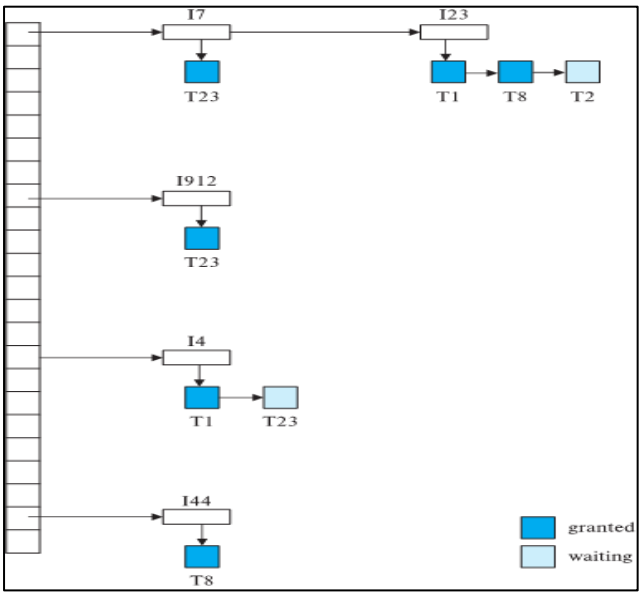
  Transactions can be serialized in the order in which they commit.

  Most databases implement rigorous two-phase locking.

Implementation of Locking

- A **lock manager** can be implemented as a separate process.
- Transactions can **send lock and unlock requests** as messages.
- The **lock manager** replies to a lock request by sending a **lock grant messages** (or a message asking the transaction to **roll back**, in case of a deadlock).
- The requesting transaction **waits until its request is answered**.
- The lock manager **maintains** an in-memory data-structure called a **lock table** to record granted locks and pending requests.

Lock Table



Dark rectangles indicate granted locks while light colored ones indicate waiting requests.

Lock table **records the type of lock** (exclusive or shared) granted or requested.

New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.

Unlock requests result in the **request being deleted**, and later requests are checked to see if they can now be granted.

If **transaction aborts, all waiting or granted requests** of the transaction are **deleted.**

TIMESTAMP-BASED PROTOCOLS

A timestamp-ordering is a method for **determining the serializability order** to select an **ordering among transactions** in advance.

Timestamps

With each transaction $\mathbf{T_1}$ in the system, we associate a **unique fixed timestamp**, denoted by TS ($\mathbf{T_1}$). This timestamp is assigned by the database system before the transaction $\mathbf{T_1}$ starts execution.

If a transaction $\mathbf{T_1}$ has been assigned timestamp TS($\mathbf{T_1}$), and a new transaction $\mathbf{T_2}$ enters the system, then TS($\mathbf{T_1}$) < TS($\mathbf{T_2}$) that is, $\mathbf{T_1}$ is older than $\mathbf{T_2}$.

Thus, if TS($\mathbf{T_1}$) < TS($\mathbf{T_2}$), then the system must **ensure that** the produced **schedule** is **equivalent to a serial schedule** in which transaction $\mathbf{T_1}$ **appears before transaction** $\mathbf{T_2}$.

To implement this scheme, we associate with each data item Q, **two timestamp values**:

• **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.

• **R-timestamp**(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are **updated** whenever a new read(Q) or write(Q) instruction is executed.

Timestamp-based protocols manage concurrent execution such that **time-stamp order = serializability order**

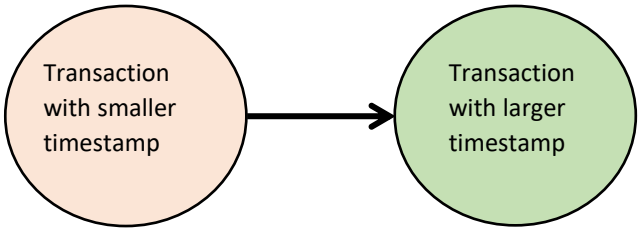
There are **two simple methods** for implementing this scheme:

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the **value of the clock** when the transaction enters the system.

2. Use a **logical counter** that is **incremented after a new timestamp** has been assigned; that is, a **transaction's timestamp is equal to the value of the counter when the transaction enters the system**.

The timestamp-ordering protocol ensures **conflict serializability**.

This is because **conflicting operations are processed in timestamp order** and the arcs in the precedence graph are as:



The protocol ensures **freedom from deadlock**, since no transaction ever waits.

DEADLOCK

A system is in a **deadlock state** if there **exists a set of transactions** such that **every transaction in the set is waiting for another transaction in the set**.

If there exists a **set of waiting transactions** {T0, T1, . . . , Tn} such that T0 is waiting for a data item that T1 holds, and T1 is waiting for a data item that T2 holds, and . . . , and Tn−1 is waiting for a data item that Tn holds, and Tn is waiting for a data item that T0 holds.

None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as **rolling back some of the transactions** involved in the deadlock.

Rollback of a transaction may be **partial**: that is, a transaction may be rolled back **to the point** where it obtained a lock **whose release resolves the deadlock**.

DEADLOCK HANDLING

There are two principal methods for dealing with the deadlock problem.

- **deadlock prevention**
- **deadlock detection** and **deadlock recovery**

Deadlock Prevention Protocol:

It ensures that the system will **never enter a deadlock state**.

Deadlock Detection and Deadlock Recovery:

Allows the system to enter a deadlock state, and then try to recover.

Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; **otherwise, detection and recovery are more efficient**.

DEADLOCK PREVENTION

There are **two approaches** to deadlock prevention.

- One approach ensures that **no cyclic waits** can occur by ordering the requests for locks, or **requiring all locks to be acquired together**.
- **Each transaction locks all its data items** before it begins execution.

The other approach is to perform **transaction rollback** whenever the wait could potentially result in a deadlock.

PREEMPTION AND TRANSACTION ROLLBACK APPROACH

The second approach for **preventing deadlocks** is to use **preemption and transaction rollbacks**.

In preemption, when a transaction $\mathbf{T_2}$ requests a lock that transaction $\mathbf{T_1}$ holds, the lock granted to $\mathbf{T_1}$ may be **preempted** by rolling back of $\mathbf{T_1}$, and granting of the lock to $\mathbf{T_2}$.

To control the preemption, we assign a **unique timestamp**, based on a **counter or on the system clock**, to **each transaction** when it begins. The system uses these timestamps only to decide whether a **transaction should wait or roll back**.

If a transaction is rolled back, it retains its **old timestamp** when restarted.

DEADLOCK PREVENTION SCHEMES

Two different deadlock-prevention schemes **using timestamps** are:

- **wait–die**
- **wound–wait**
- Another approach is **lock timeouts**.

WAIT–DIE SCHEME

The **wait–die** scheme is a **non-preemptive** technique.

- When transaction $\mathbf{T_1}$ requests a data item currently held by $\mathbf{T_2}$, $\mathbf{T_1}$ is allowed **to wait** only if it has a **timestamp smaller** than that of $\mathbf{T_2}$ (that is, $\mathbf{T_1}$ is older than $\mathbf{T_2}$). Otherwise, $\mathbf{T_1}$ is **rolled back** (dies).

Example:

Transactions **T1, T2, and T3** have timestamps **5, 10, and 15**, respectively.

If **T1** requests a data item held by **T2**, then **T1** will wait. (**smaller timestamp)**.

If **T3** requests a data item held by **T2**, then **T3** will be rolled back. (**larger timestamp).**

WOUND–WAIT SCHEME

The **wound–wait** scheme is a **preemptive** technique & **counterpart** to the **wait–die** scheme.

- When transaction T_1 requests a data item currently held by T_2 , T_1 is allowed to wait only if it has a **timestamp larger** than that of T_2 (that is, T_1 is younger than T_2). Otherwise, T_2 is rolled back (T_2 is wounded by T_1).

Example:

Transactions **T1, T2, and T3** have timestamps **5, 10, and 15**, respectively.

If **T1** requests a data item held by **T2**, then the data item will be preempted from **T2**, and **T2** will be rolled back (wound).

If **T3** requests a data item held by **T2**, then**T3** will **wait**.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

LOCK TIMEOUTS SCHEME

Another simple approach to deadlock prevention is based on **lock timeouts**.

In this approach, a transaction that has **requested a lock waits for at most a specified amount of time**. If the lock has **not been granted** within that time, the transaction is said to **time out**, and it **rolls itself back and restarts**.

If there was in fact a deadlock, one or more transactions involved in the deadlock will **time out and roll back**, allowing the others to proceed.

DEADLOCK DETECTION AND RECOVERY

If a system **does not employ some protocol** that ensures deadlock freedom, then a **detection and recovery scheme must be used**.

An algorithm that **examines** the state of the **system** is **invoked periodically** to determine whether a **deadlock has occurred**.

If one has, then the system **must attempt to recover** from the deadlock.

To recover from the deadlock, **the system must**:

- Maintain information about the **current allocation of data items to transactions**, as well as **any outstanding data item requests**.
- **Provide an algorithm** that uses this information to **determine whether the system has entered a deadlock state.**
- **Recover from the deadlock** when the detection algorithm determines that a **deadlock exists**.

DEADLOCK DETECTION

Deadlocks in the system can be represented as **directed graph** called as **wait-for graph**.

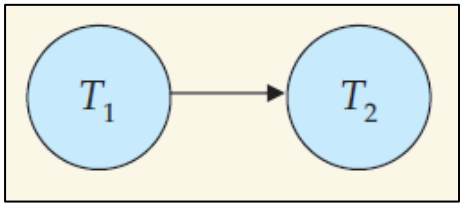
This graph consists of a pair G = (V, E) where

V is a set of **vertices** and

E is a set of **edges**.

The **set of vertices** consists of all the **transactions** in the system.

The **wait-for graph** for **transactions T_1 and T_2** with edge between them can be represented as



DEADLOCK DETECTION WITH WAIT-FOR GRAPH

Each element in the **set E of edges** is an **ordered pair** as **$T_1 \rightarrow T_2$**.

If **$T_1 \rightarrow T_2$** is in set **E**, then there is a **directed edge** from transaction **T_1** to **T_2** , implying that transaction **T_1** is **waiting for** transaction **T_2 to release a data item** that it needs.

Example: wait-for graph

When transaction **T_1** requests a data item currently being held by transaction **T_2**, then the **edge $T_1 \rightarrow T_2$** is **inserted in the wait-for graph** and This **edge is removed** only when transaction **T_2** is no longer holding a data item needed by transaction **T_1**.

A deadlock exists in the system **if and only if** the **wait-for graph** contains a **cycle**.

Each transaction involved in the cycle **is said to be deadlocked**.

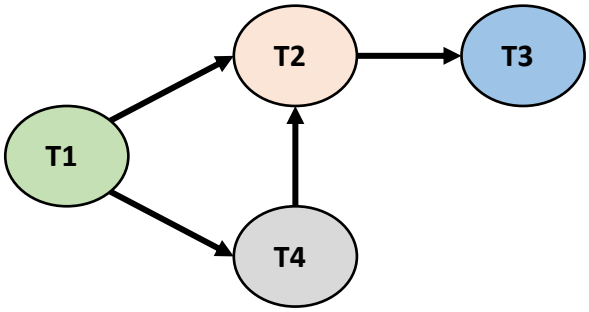
To detect deadlocks, the system needs to maintain the wait-for graph, and **periodically to invoke an algorithm** that searches for a **cycle in the graph**.

A deadlock exists in the system **if and only if** the **wait-for graph** contains a **cycle**.

Each transaction involved in the cycle **is said to be deadlocked**.

To detect deadlocks, the system needs to maintain the wait-for graph, and **periodically to invoke an algorithm** that searches for a **cycle in the graph**.

Consider the wait-for graph with transactions T1, T2, T3 & T4 as

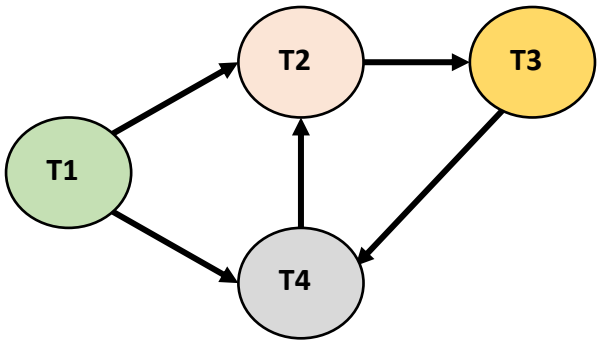


The graph depicts the following **situation**:

- Transaction **T1 is waiting** for transactions **T2 and T4**.
- Transaction **T4 is waiting** for transaction **T2**.
- Transaction **T2 is waiting** for transaction **T3**.

Since the graph has **no cycle**, the system is **not in a deadlock state**.

Suppose now that transaction **T3 is requesting an item held by T4**. The edge **T3 → T4** is **added to the wait-for graph**, resulting in the **new system state** in figure as,



The graph depicts the following situation:

- Transaction **T2 is waiting** for transactions T3.
- Transaction **T3 is waiting** for transaction T4.
- Transaction **T4 is waiting** for transaction T2.

This time, the graph contains the **cycle**: **T2 →T3 →T4 →T2.**

Thus, transactions **T2, T3, and T4** are all **deadlocked**.

RECOVERY FROM DEADLOCK

When a **detection algorithm** determines that a **deadlock exists**, the system must **recover** from the deadlock.

The most common solution is to **roll back one or more transactions** to break the deadlock. **Three actions** need to be taken:

1. **Selection of a victim**
2. **Rollback**
3. **Starvation**

1. SELECTION OF A VICTIM -

Given a set of deadlocked transactions, we must determine **which transaction** (or transactions) **to roll back** to **break the deadlock**.

We should roll back those transactions that will incur the **minimum cost**.

Many factors may **determine the cost of a rollback**, including:

- **How long** the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
- **How many data items** the transaction has **used**.
- **How many more data items** the transaction **needs** for it to complete.
- **How many transactions will be involved** in the rollback.

2. ROLLBACK-

Once we have decided that a particular transaction must be **rolled back**, we must determine **how far this transaction should be rolled back**.

There are two ways:

- **total rollback**
- **partial rollback**

1. **Total rollback**: It is simplest solution. It **aborts** the transaction and **then restart** it.

2. **Partial rollback:** It rolls back the transaction only as far as necessary **to break the deadlock**.

Such **partial rollback** requires the system to **maintain additional information** about the **state of all the running transactions**.

Specifically, the **sequence of lock requests/grants and updates** performed by the transaction **needs to be recorded**.

The deadlock detection mechanism should decide **which locks** the selected transaction needs to release in order to break the deadlock.

The selected transaction must be **rolled back to the point** where it obtained the first of these locks, undoing all actions it took after that point.

The recovery mechanism must be **capable of resuming execution after a partial rollback**.

3. STARVATION-

In a system where the selection of victims is based primarily on **cost factors**, it may happen that the **same transaction** is always **picked as a victim**.

As a result, this transaction never completes its designated task, thus there is **starvation**.

We must ensure that a transaction can be picked as a **victim only a (small) finite number of times.**

The most common solution is to include the **number of rollbacks in the cost factor**.

RECOVERY SYSTEM (FROM FAILURE)

A computer system is **subject to failure** from a variety of causes:

- disk crash,
- power outage,
- software error,
- any miss-happening in the machine room, or
- any intentional act of disruption.

- In any failure, **information may be lost**. Therefore, the database system must take actions **in advance** to ensure that the atomicity and durability properties of transactions are preserved.

- An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.
- The recovery scheme must also provide **high availability**; that is, it must **minimize** the time for which the **database is not usable** after a failure.

Failure Classification

Following are the **various types of failure** that may occur in a system:

1. Transaction failure -

There are **two types of errors** that may **cause a transaction to fail**:

- **Logical error-**

The transaction can **no longer continue with its normal execution** because of some internal condition, such as **bad input, data not found, overflow, or resource limit exceeded**.

- **System error-**

The system has entered an **undesirable state** (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

2. System crash-

- There is a **hardware malfunction, or a bug in the database software or the operating system**, that causes the loss of the content of volatile storage, and brings transaction processing to a halt.

The content of **nonvolatile storage remains intact, and is not corrupted.**

3. Disk failure-

- A **disk block loses its content** as a result of either a **head crash or failure during a data-transfer operation**.
- Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

STORAGE STRUCTURE

The various data items in the database may be **stored and accessed** in a number of **different storage media.**

The different storage media can be **distinguished** by their **relative speed, capacity, and resilience to failure**.

There are **three categories of storage**:

1. **Volatile storage**
2. **Non-volatile storage**
3. **Stable storage**

Stable storage plays a **critical role in recovery algorithms**.

Categories of storage media

1. Volatile storage:

- **Does not survive** system crashes
- **Examples**: main memory, cache memory

2. Nonvolatile storage:

- **Survives system crashes**
- **Examples**: disk, tape, flash memory, non-volatile RAM
- But **may still fail, losing data**

3. Stable storage:

- A mythical form of storage that **survives all failures**
- To implement stable storage, we need to **replicate the needed information in several nonvolatile storage media** (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

RECOVERY ALGORITHMS

- Suppose transaction T1 transfers ₹50 from account A to account B.
- It requires **two updates**-
    - subtract ₹50 from account A and
    - add ₹50 to account B.
- Transaction T1 requires **updated values of A and B will be the output** of database.
- A **failure** may occur **after one of these modifications** have been made but **before both of them are made permanent.**
- **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 algorithm has 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.**

RECOVERY TECHNIQUES

There are **two recovery techniques** as

- **Shadow-paging**
- **Log based recovery**

SHADOW COPY & SHADOW PAGING

Shadow-copy scheme:

In the **shadow-copy** scheme, a transaction that wants to update the database first creates a **complete copy of the database**. All **updates are done on the new database copy**, leaving the original copy, the **shadow copy**, untouched.

If at any point the **transaction has to be aborted**, the system merely **deletes the new copy**. The old copy of the database has **not been affected**.

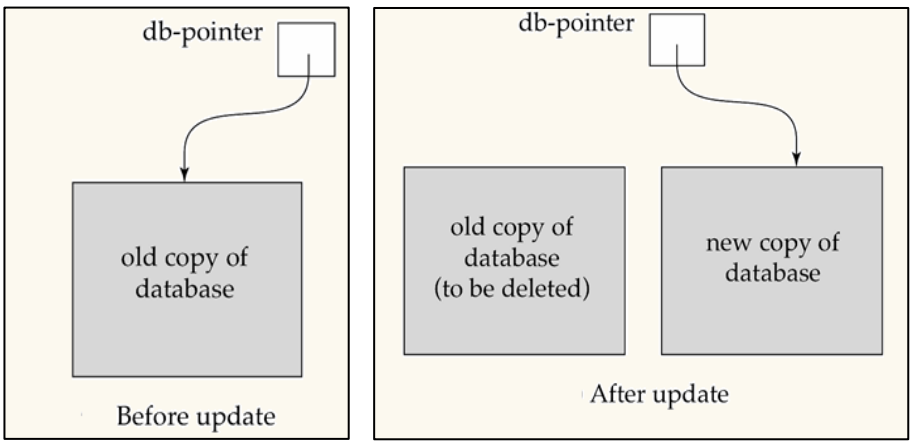
The current copy of the database is identified by a pointer, called **db-pointer**, which is stored on disk.

If the **transaction partially commits** (that is, executes its final statement), it is **committed as follows**:

- First, the operating system is asked to make sure that **all pages of the new copy** of the database have been **written out to disk**.
- After the operating system has written all the pages to disk, **the database system updates the pointer db-pointer to point to the new copy of the database**; the new copy then becomes the current copy of the database.
- The **old copy** of the database is then **deleted**.
- The transaction is said to have been committed at the point where the **updated db-pointer is written to disk.**

  Shadow copy schemes are commonly used by **text editors** (saving the file is equivalent to transaction commit, while quitting without saving the file is equivalent to transaction abort).

  Shadow copying can be **used for small databases**, but copying a **large database** would be **extremely expensive**.



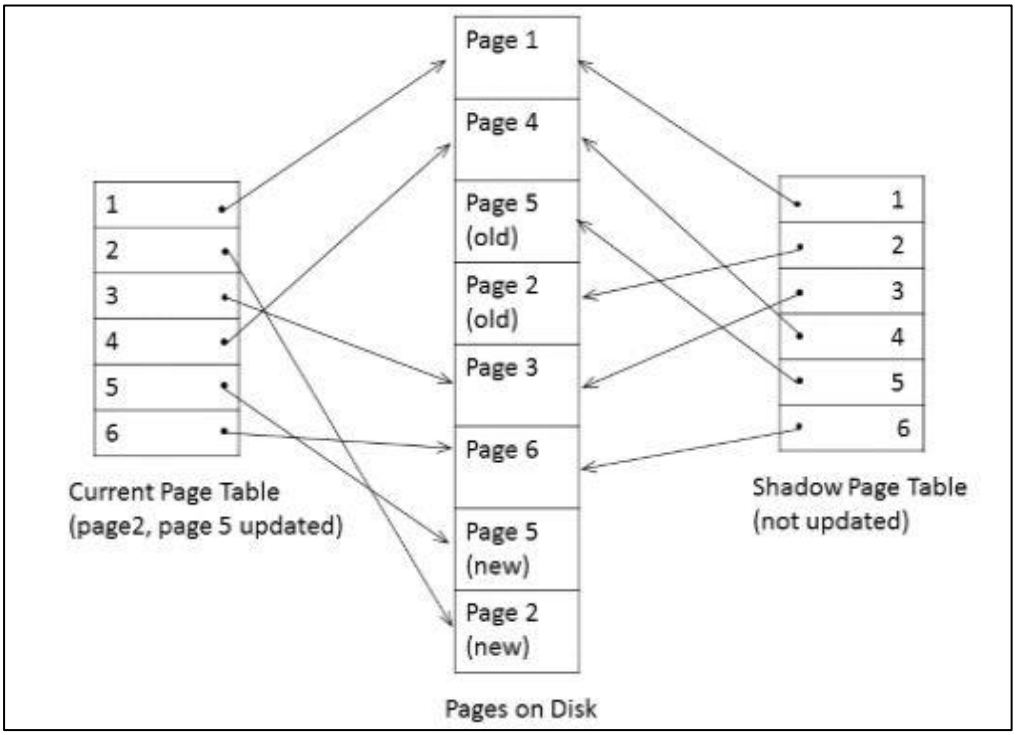
Shadow copy

Shadow Paging

A **variant of shadow copying**, called **shadow-paging**, reduces copying as follows:

- The scheme uses a **page table containing pointers to all pages**; the page table itself and all updated pages are copied to a new location.
- **Any page which is not updated by a transaction is not copied**, but instead the new page table just stores a pointer to the original page.
- When a transaction commits, it atomically **updates the pointer to the page table**, which acts as db-pointer, to point to the new copy.
- Shadow paging unfortunately does not work well with **concurrent transactions** and is **not widely used in databases**.

Shadow Paging can be illustrated as



LOG BASED RECOVERY

Log or Log records:

The most widely used structure for recording database modifications is the **log**.

The log is a sequence of **log records**, recording **all the update activities** in the database.

An **update log record** describes a single database write.

An update log record has **following fields**:

- **Transaction identifier**, which is the **unique identifier of the transaction** that performed the write operation.
- **Data-item identifier**, which is the **unique identifier of the data item written**. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- **Old value**, which is the **value of the data item prior to the write**.
- **New value**, which is the **value that the data item will have after the write**.

An update log record is represented as

< **Ti, Xj, V1, V2** >

Log record indicates that **Transaction Ti** has performed a **write operation** on **data item Xj ,** where **V1 is the value before the write**, and **V2 is the value after the write**.

Other types of special log records are:

• **<Ti start>** - Transaction Ti has started.

• **<Ti commit>** - Transaction Ti has committed.

• **<Ti abort>** - Transaction Ti has aborted.

Database log corresponding to transactions T_0 and T_1 can be shown as

<T_0 start>
<T_0 , A, 950>
<T_0 , B, 2050>
<T_0 commit>
<T_1 start>
<T_1 , C, 600>
<T_1 commit>

Whenever a transaction performs a **write**, it is essential that the log record for that write be **created and added to the log**, before the database is **modified**.

Once a **log record exists, we can output the modification to the database** if that is desirable.

Log	Database
$<T_0$ start>	
$<T_0, A, 950>$	
$<T_0, B, 2050>$	
$<T_0$ commit>	
	$A = 950$
	$B = 2050$
$<T_1$ start>	
$<T_1, C, 600>$	
$<T_1$ commit>	
	$C = 600$

RECOVERY AND ATOMICITY

When a **DBMS recovers from a crash**, it should **maintain the following** -

- It should **check the states of all the transactions**, which were being executed.
- A transaction may be in the middle of some operation; the **DBMS must ensure the atomicity of the transaction** in this case.
- It should check whether the **transaction can be completed now** or it **needs to be rolled back.**

No transactions would be allowed to leave the DBMS in an **inconsistent state**.

MAINTAINING ATOMICITY OF TRANSACTION

There are **two types of techniques**, which can help a DBMS in recovering as well as **maintaining the atomicity of a transaction** -

- **Maintaining the logs** of each transaction, and **writing them onto some stable storage** before actually modifying the database.
- **Maintaining shadow paging**, where the changes are done on a volatile memory, and later, the **actual database is updated**.

CHECKPOINTS

When a **system crash occurs**, we must **consult the log** to determine those transactions that need to be **redone** and those that need to be **undone**.

In principle, we need **to search the entire log** to determine this information.

There are **two major difficulties** with this approach:

1. The search process is **time-consuming**.

2. Most of the transactions that, according to algorithm, need to be **redone** have already written their updates into the database.

Although redoing them will cause no harm, it will nevertheless **cause recovery to take longer**.

To **reduce these types of overhead**, we introduce **checkpoints**.

A checkpoint is performed as follows:

1. Output onto stable storage all log records currently residing in main memory.

2. Output **to the disk** all modified buffer blocks.

3. Output **onto stable storage** a log record of the form **<checkpoint L>**,

where **L is a list of transactions active** at the time of the checkpoint.

Transactions **are not allowed to perform any update actions**, such as writing to a buffer block or writing a log record, **while a checkpoint is in progress.**

RECOVERY WITH CHECKPOINTS

The presence of a <checkpoint L> record in the log allows the system to **streamline its recovery** procedure.

Example:

Consider a transaction Ti that completed **prior to the checkpoint**. For such a transaction, the <Ti commit>record (or < Ti abort> record**) appears in the log before the <checkpoint> record**.

Any database modifications made by Ti **must have been written to the database** either **prior to the checkpoint** or **as part of the checkpoint itself**.

Thus, at recovery time, there is **no need to perform a redo operation on Ti** .

After a system **crash has occurred**, the **system examines the log to find the last <checkpoint L>** record (this can be done by searching the log backward, from the end of the log, until the first <checkpoint L> record is found).

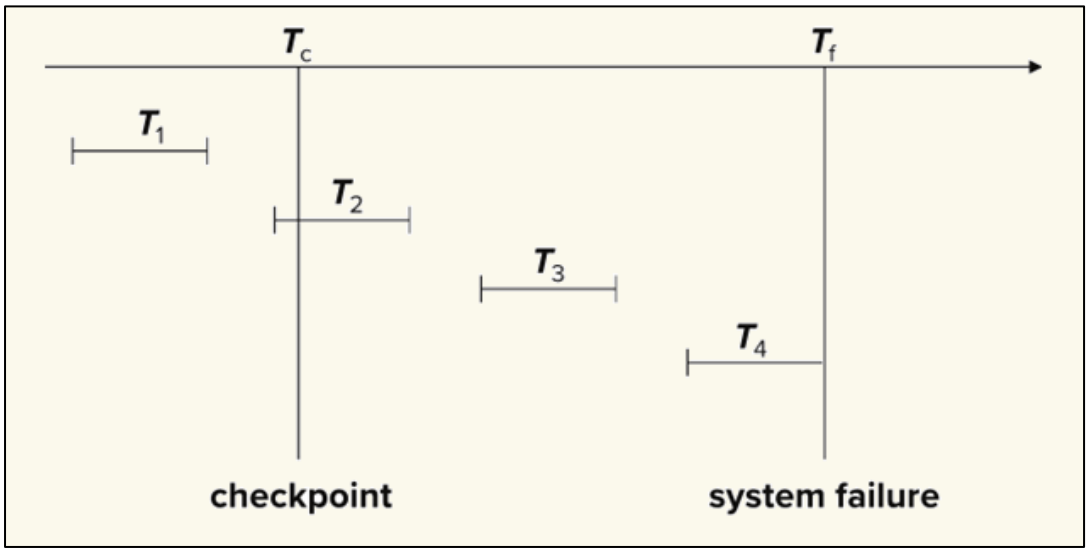
When a system with concurrent transactions **crashes and recovers**, it behaves in the following manner –

- The recovery system **reads the logs backwards** from the end to the last checkpoint.
- It maintains two lists, an **undo-list and a redo-list**.
- If the recovery system sees a **log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>**, it puts the transaction in the **redo-list**.
- If the recovery system sees a **log with <T_n, Start> but no commit or abort log** found, it puts the transaction in **undo-list**.

All the transactions in the **undo-list are then undone** and their logs are removed.

All the transactions in the **redo-list and their previous logs are removed and then redone** before saving their logs.

Example-Recovery with checkpoints



In case of recovery above system behaves in the following manner −

- Transaction T_1 can be **ignored** (**updates already output to disk** due to checkpoint)
- Transactions T_2 and T_3 **redone (redo** sets data item specified to the **new value)**.
- Transaction T_4 **undone (undo** sets data item specified to the **old value).**

Reference: Database System Concepts, Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Sixth Edition Mcgraw-Hill Publication