

# DATABASE DESIGN AND MANAGEMENT

## UNIT I: CONCEPTUAL DATA MODELING

**Database environment – Database system development lifecycle – Requirements collection – Database design --Entity-Relationship model – Enhanced-ER model – UML class diagrams.**

### INTRODUCTION

**Data:** Raw fact i.e unprocessed data.

**Eg:** Suresh, 25, Chennai

**Information:** Processed data.

**Eg:** The age of suresh is 25

### **Database:**

A Database is a collection of related data organized in a way that data can be easily accessed, managed and updated. Any piece of information can be a data, for example name of your school. Database is actually a place where related piece of information is stored and various operations can be performed on it.

**Database Management System (DBMS):** The software which is used to manage database is called Database Management System (DBMS). For Example, MySQL, Oracle etc. are popular commercial DBMS used in different applications.

DBMS allows users the following tasks:

**Data Definition:** It helps in creation, modification and removal of definitions that define the organization of data in database.

**Data Updation:** It helps in insertion, modification and deletion of the actual data in the database.

**Data Retrieval:** It helps in retrieval of data from the database which can be used by applications for various purposes.

**User Administration:** It helps in registering and monitoring users, enforcing data security, monitoring performance, maintaining data integrity, dealing with concurrency control and recovering information corrupted by unexpected failure.

### **Database Applications:**

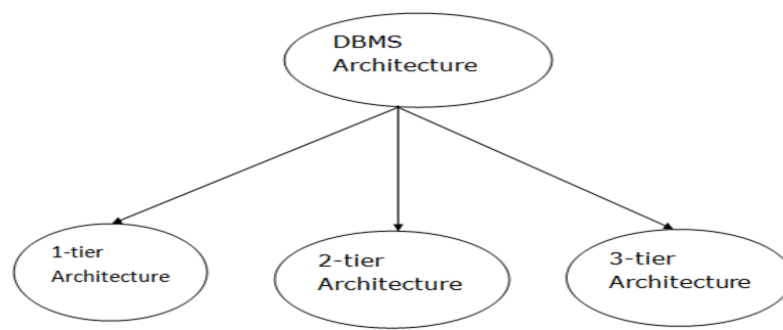
1. Banking: all transactions
2. Airlines: reservations, schedules
3. Universities: registration, grades
4. Sales: customers, products, purchases
5. Online retailers: order tracking, customized recommendations
6. Manufacturing: production, inventory, orders, supply chain

7. Human resources: employee records, salaries, tax deductions

### DBMS Architecture

- The DBMS design depends upon its architecture. The basic client/server architecture is used to deal with a large number of PCs, web servers, database servers and other components that are connected with networks.
- The client/server architecture consists of many PCs and a workstation which are connected via the network.
- DBMS architecture depends upon how users are connected to the database to get their request done.

### Types of DBMS Architecture



Database architecture can be seen as a single tier or multi-tier. But logically, database architecture is of two types like: **2-tier architecture** and **3-tier architecture**.

### 1-Tier Architecture

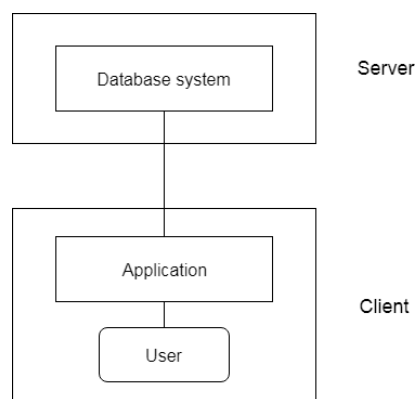
- In this architecture, the database is directly available to the user. It means the user can directly sit on the DBMS and uses it.
- Any changes done here will directly be done on the database itself. It doesn't provide a handy tool for end users.
- The 1-Tier architecture is used for development of the local application, where programmers can directly communicate with the database for the quick response.
- **Tier Architecture** in DBMS is the simplest architecture of Database in which the client, server, and Database all reside on the same machine. A simple one tier architecture example would be anytime you install a Database in your system and access it to practice SQL queries. But such architecture is rarely used in production.



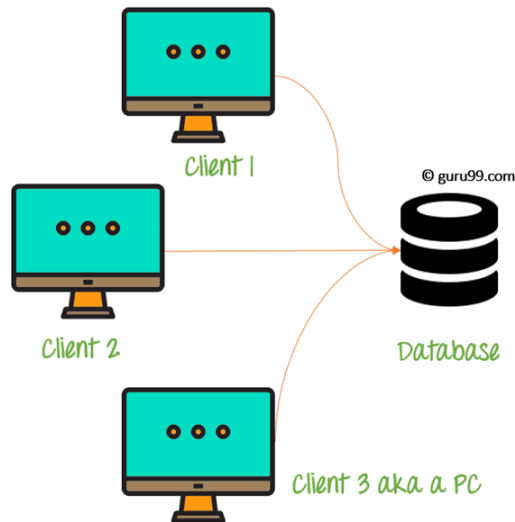
## Single Tier Architecture

### 2-Tier Architecture

- The 2-Tier architecture is same as basic client-server. In the two-tier architecture, applications on the client end can directly communicate with the database at the server side. For this interaction, API's like: **ODBC**, **JDBC** are used.
- The user interfaces and application programs are run on the client-side.
- The server side is responsible to provide the functionalities like: query processing and transaction management.
- To communicate with the DBMS, client-side application establishes a connection with the server side.
- A 2 Tier Architecture in DBMS is a Database architecture where the presentation layer runs on a client (PC, Mobile, Tablet, etc.), and data is stored on a server called the second tier. Two tier architecture provides added security to the DBMS as it is not exposed to the end-user directly. It also provides direct and faster communication.



**Fig: 2-tier Architecture**



In the above 2 Tier client-server architecture of database management system, we can see that one server is connected with clients 1, 2, and 3.

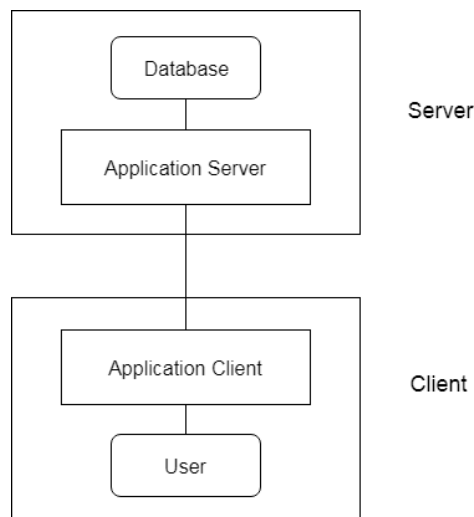
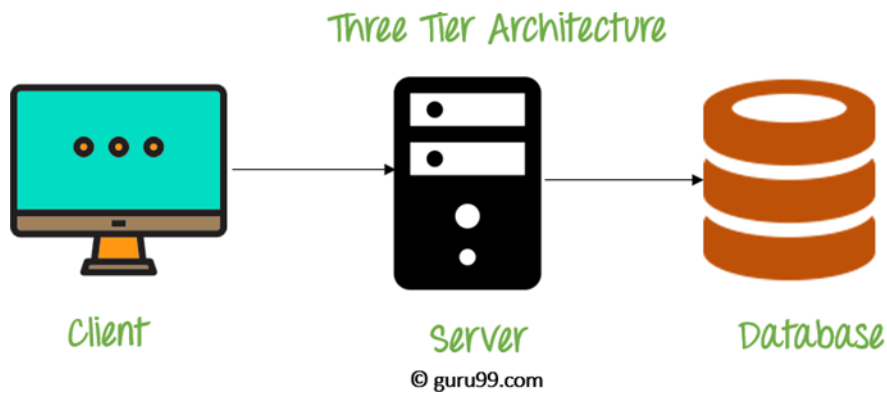
### 3-Tier Architecture

- The 3-Tier architecture contains another layer between the client and server. In this architecture, client can't directly communicate with the server.
- The application on the client-end interacts with an application server which further communicates with the database system.
- End user has no idea about the existence of the database beyond the application server. The database also has no idea about any other user beyond the application.
- The 3-Tier architecture is used in case of large web application.
- 3-Tier database Architecture design is an extension of the 2-tier client-server architecture. A 3-tier architecture has the following layers:
  - Presentation layer (your PC, Tablet, Mobile, etc.)
  - Application layer (server)
  - Database Server

The Application layer resides between the user and the DBMS, which is responsible for communicating the user's request to the DBMS system and send the response from the DBMS to the user. The application layer(business logic layer) also processes functional logic, constraint, and rules before passing data to the user or down to the DBMS.

The goal of Three Tier client-server architecture is:

- To separate the user applications and physical database
- To support DBMS characteristics
- Program-data independence
- Supporting multiple views of the data



**Fig: 3-tier Architecture**

### Three schema Architecture

- The three schema architecture is also called ANSI/SPARC architecture or three-level architecture.
- This framework is used to describe the structure of a specific database system.
- The three schema architecture is also used to separate the user applications and physical database.
- The three schema architecture contains three-levels. It breaks the database down into three different categories.

### **DATABASE ENVIRONMENT**

One of the primary aims of a database is to supply users with an abstract view of data, hiding a certain element of how data is stored and manipulated. Therefore, the starting point for the design of a database should be an abstract and general description of the information needs of the organization that is to be represented in the database. And hence you will require an environment to store data and make it work as a database.

## VIEWS OF DATA/DATA ABSTRACTION:

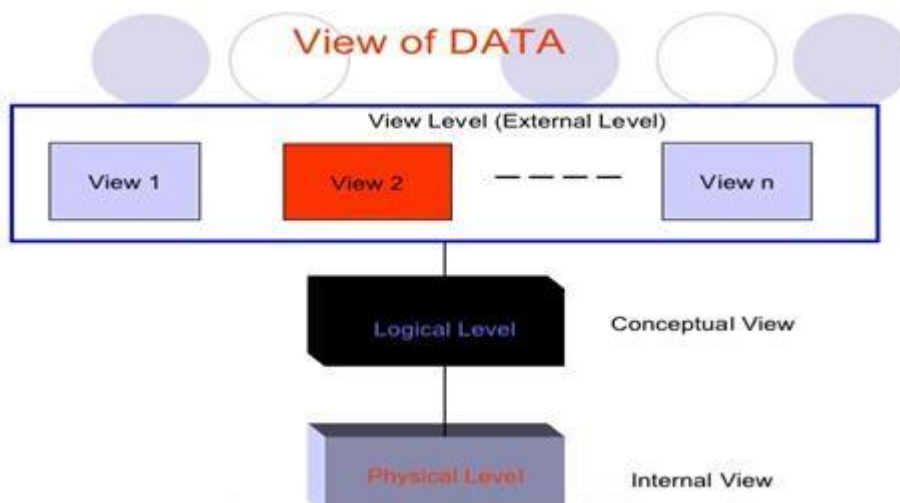
A major purpose of a database system is to provide users with an abstract view of data. I.e., the system hides certain details of how the data are stored and maintained.

Three Schema Architecture:

Separates the user applications and physical database. Schemas can be defined in three levels:

### (i) Internal Level:

- It has an internal schema which describes physical storage structure of the database.
- How the data are actually stored uses physical model
- describes the complete details of data storage and access paths for the database. **e.g., customer**



### (ii) Conceptual Level:

- It has an conceptual schema which describes the structure of whole database
- What data are stored and what relationships exist among data.
- Uses high level or implementational data model.
  - Hides the details of physical storage structures and describes datatypes, relationships, operations and constraints.

e.g:

```
typecustomer = record
  customer_id : string;
  customer_name : string;
  customer_street : string;
  customer_city : string;
end;
```

**(iii) External or View Level:**

- includes a number of external schemas or views.
- Each external schema describes the part of the database and hides the rest.
- Uses high level or implementation data model.

such as an employee's salary.

**Instances and Schemas**

The state of the database changes over time as information is inserted and deleted by the users.

**Instance:**

The collection of information stored in the database at a particular moment is called Instance of the database.

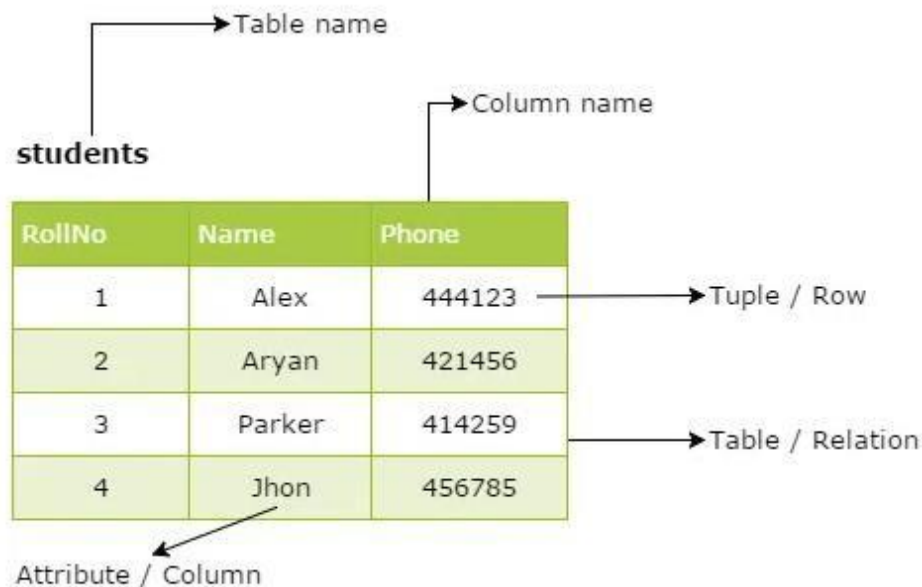
**Schema:**

The overall design of the database is called the database schema.

When you talk about the database, you must distinguish between the database schema, which is the logical blueprint of the database, and the database instance, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming language notion of a variable. In contrast, the concept of a relation schema corresponds to the programming languages' notion of the type definition. In other words, a database schema is a skeletal structure that represents the logical view of the complete database. It describes how the data is organized and how the relations among them are associated and formulates all the constraints that are to be applied to the data.

In general, a relation schema consists of a directory of attributes and their corresponding domain.

**Some Common Relational Model Terms**



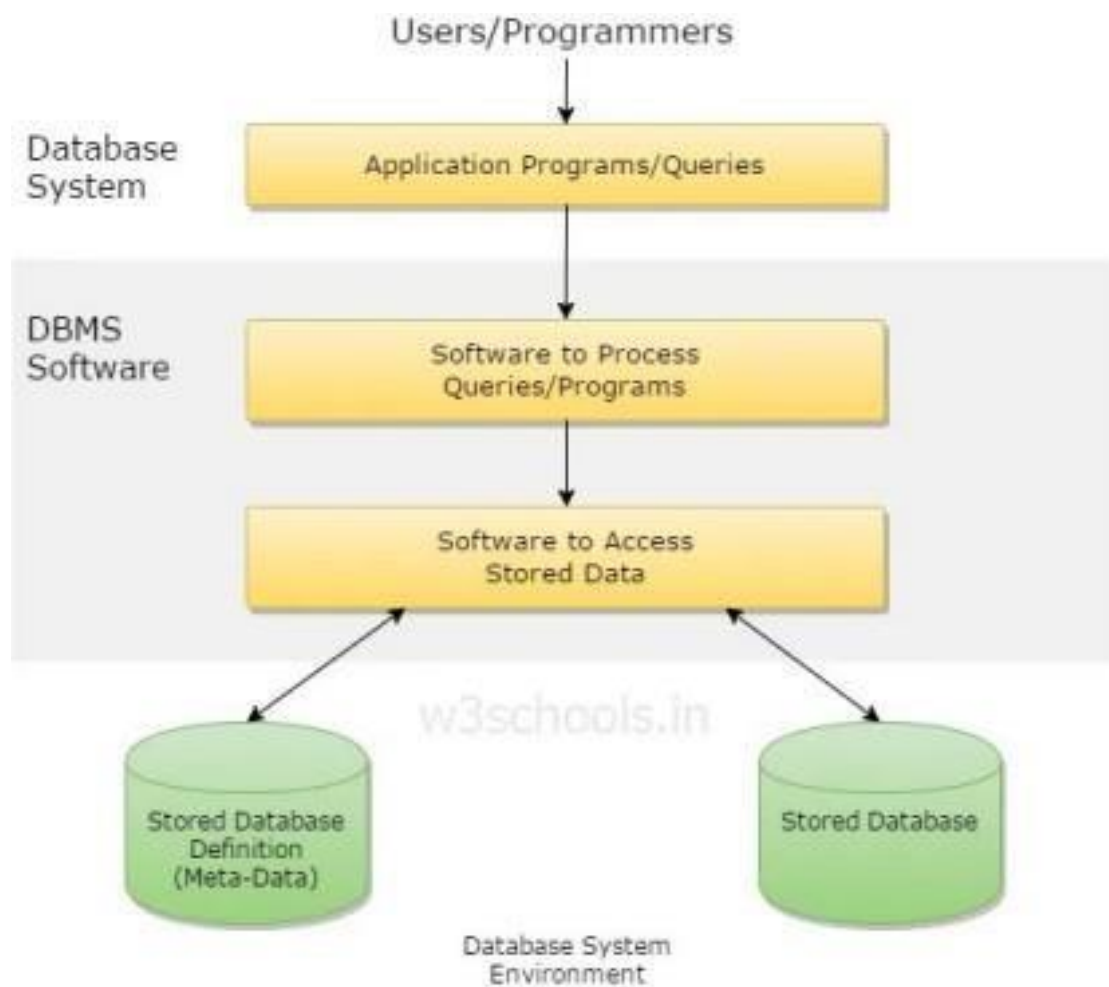
Relational Model Terms

## Some Common Relational Model Terms

- **Relation:** A relation is a table with columns and rows.
- **Attribute:** An attribute is a named column of a relation.
- **Domain:** A domain is the set of allowable values for one or more attributes.
- **Tuple:** A tuple is a row of a relation.

A database schema can be divided broadly into two categories –

- **Physical Database Schema** – This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
- **Logical Database Schema** – This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.



A database environment is a collective system of components that comprise and regulates the



group of data, management, and use of data, which consist of software, hardware, people, techniques of handling database, and the data also.

Here, the hardware in a database environment means the computers and computer peripherals that are being used to manage a database, and the software means the whole thing right from the operating system (OS) to the application programs that include database management

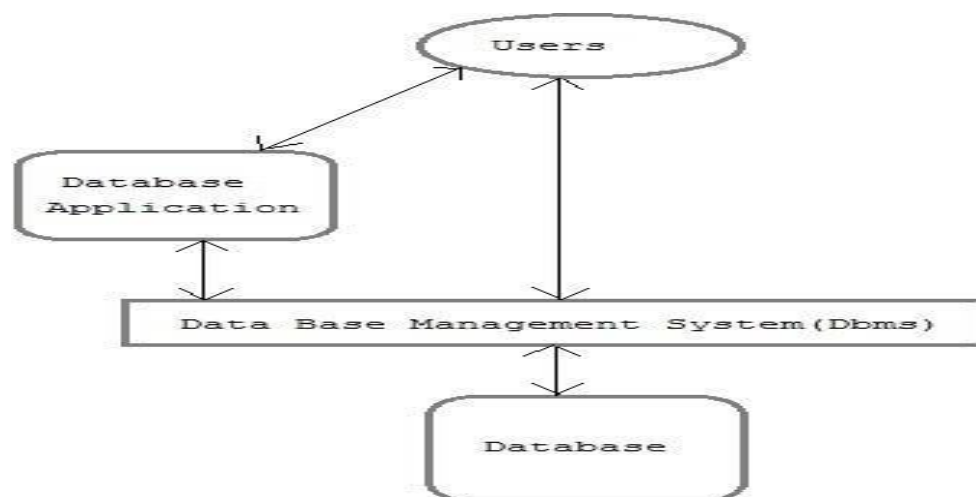
software like M.S. Access or SQL Server. Again the people in a database environment include those people who administrate and use the system. The techniques are the rules, concepts, and instructions given to both the people and the software along with the data with the group of facts and information positioned within the database environment.

### **Components of Database System Environment**

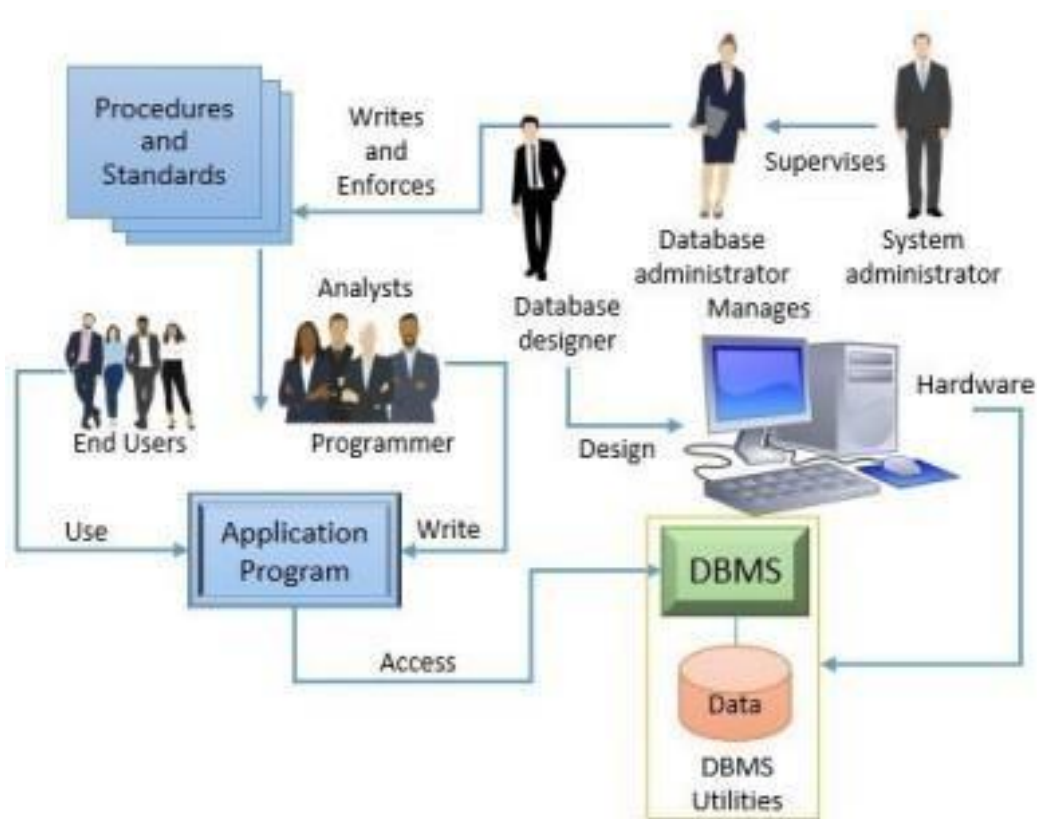
Every system environment is made up of certain components that help the system to get organized and managed. Even the database system environment is made up of the following components:

### **Componentes of Database System**

The database system can be divided into four components.



- Users: Users may be of various type such as DB administrator, System developer and End users.
- Database application : Database application may be Personal, Departmental, Enterprise and Internal
- DBMS: Software that allow users to define, create and manages database access, Ex: MySQL, Oracle etc.
- Database: Collection of logical data.



### 1. Hardware

The hardware component of the database system environment includes all the physical devices that comprise the database system. It includes storage devices, processors, input and output devices, printers, network devices and many more.

### 2. Software

The software component of the database environment includes all the software that we require to access, store and regulate the database. Like operating systems, DBMS and application programs and utilities. The operating system invokes computer hardware, and let other software runs. DBMS software controls and regulates the database. The application program and utilities access the database and if required you can even manipulate the database.

### 3. People

If talk of the people component then it will include all the people who are related to the database. There may be a group of people who will access the database just to resolve their

queries i.e. end-user, there may be people that are involved in designing the database i.e. database designer.

There are four distinct types of people that participate in the DBMS environment: data and database administrators, database designers, application developers, and the end-users.

Some may be involved in designing the applications that will have an interface through which data entry is possible i.e. database programmer and analyst and some may also be there to monitor the database i.e. database administrator.

#### **4. Procedures**

The procedure component of the database environment is nothing but the function that regulates and controls the use of the database.

#### **5. Data**

Data component include a collection of related data which are the known fact that can be recorded and it has an implicit meaning in the databsae.

##### **System Utilities**

Database system utilities are the tools that can be used by the database system administrator to control and manage the database system.

##### **System Utilities**

Database system utilities are the tools that can be used by the database system administrator to control and manage the database system.

##### **1. Loading Utility**

Loading database utility helps in loading the database file into the database. It efficiently reformats the current format of data files to the format that is required by the destination database file structure. Some loading programs or tools are specially designed for loading data from one DBMS to another.

If you provide source database storage description and target database storage description to these loading tools then it will automatically reformat the data files to target database storage description.

##### **2. Backup Utility**

The backup utility in the database environment helps in creating a backup copy of the entire database. Generally, the entire data of the database is copied to mass storage and we refer to it as a backup copy. This backup copy can be used when there is a system failure or storage of your system is corrupted.

You can always choose incremental backups which only record the changes from the previous backup. Though the incremental backup requires a more complex algorithm it saves more space as compared to regular backup.

##### **3. Database Storage Reorganization Utility**

Sometimes we need to relocate the set of database files to a different location. The database storage reorganization utility helps to relocate and organize the database files to a different location and it also produces a new access path to access the files from its new location.

##### **4. Performance Monitoring Utility**

Performance monitoring utility monitors the usage of the database by its user and provides statistics for the same to the database administrator (DBA). The statistics provided by the utility helps the DBA to decide whether it is required to reorganize the data files, whether

there is a need to add new indexes or not, whether some indexes to the files must be dropped to improve the performance of the database system.

There are more utilities in the database environment that help in sorting the database file on some basis, handling data compression on the large databases, monitoring the user's access to the database and many more.

## **2. DATABASE SYSTEM DEVELOPMENT LIFECYCLE**

### **Explanation:1**

As a database system is a primary element of the more extensive organization-wide information system, the database system development life cycle is inherently connected with the life cycle of the information system. The stages of the database system development lifecycle are shown in the figure below:

#### **Database System Development Lifecycle**

- Database planning
- System definition
- Requirements collection and analysis
- Database design (conceptual, logical, physical)
- DBMS selection (optional)
- Application design
- Prototyping (optional)
- Implementation
- Data conversion and loading
- Testing
- Operational maintenance

#### **Database Planning – Mission Statement**

An essential first step in database planning is to define the mission statement for the database system. The mission statement describes the primary aims of the database system. Those are driving the database project within the organization that generally defines the mission statement. A mission statement helps to simplify the purpose of the database system and provide a clearer path towards the efficient and effective creation of the required database system.

It is the management activities that permit the stages of the database system development life cycle to be realized as efficiently and effectively as possible.

Database planning must be integrated with the overall IS strategy of the organization.

There are three main issues involved in formulating an IS strategies which are:

- Identification of enterprise plans and goals with the subsequent purpose of information systems requirements
- Evaluation of current information systems to find out existing strengths and weaknesses
- Appraisal of IT opportunities that might yield aggressive advantage

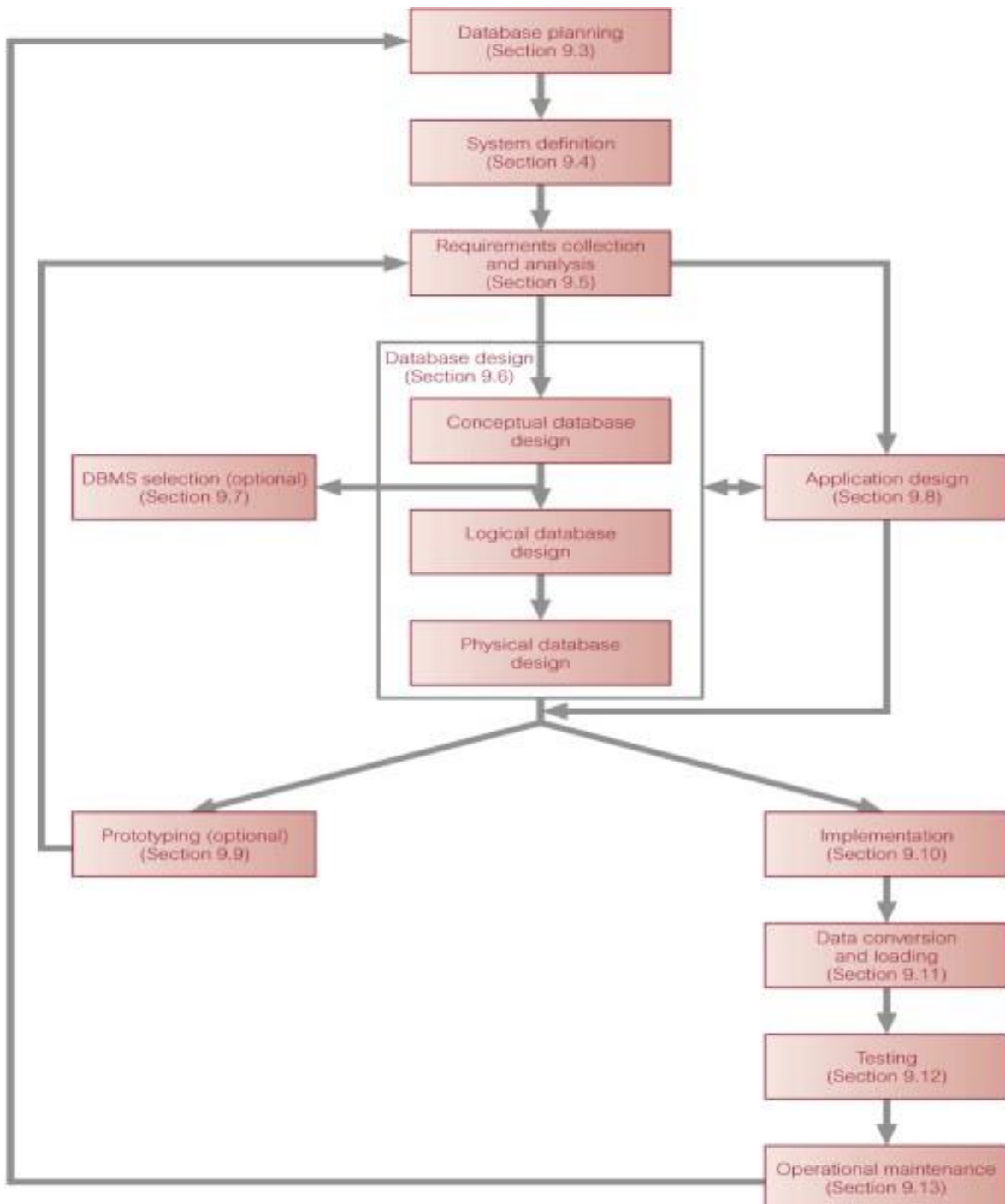
**Ex.**

-The purpose of our HW database system is to maintain the data that is used to support hotel room rentals

- Once mission statement is defined, mission objectives are defined which should identify a particular task that the database must support.

**Ex.**

To maintain (insert, update, delete) data on the hotels, rooms, guests, and bookings.

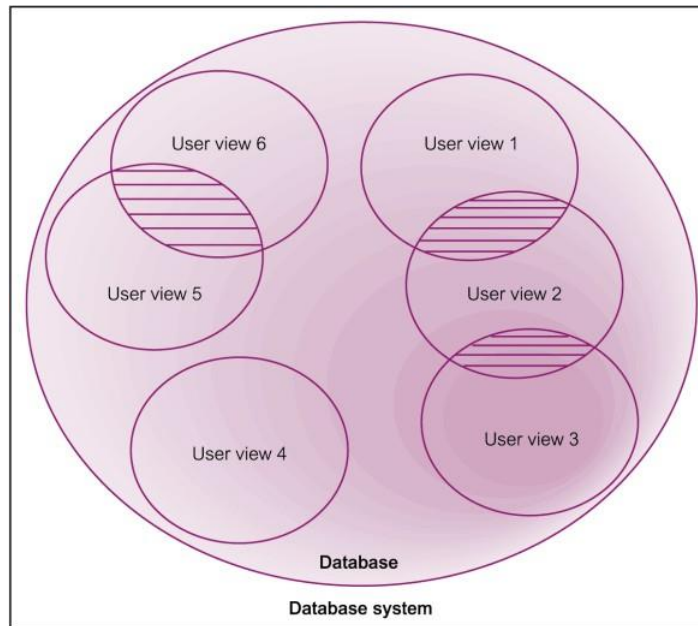


**System Definition**

- Describes scope and boundaries of database system and the major user views.
- User view defines what is required of a database system from the perspective of:

- a particular job role (such as Manager or Supervisor) or
- enterprise application area (such as marketing, personnel, etc.).

### Representation of a Database System with Multiple User View



### Requirements Collection and Analysis

- Ø Get user requirements - collect and analyze information about the part of organization to be supported by the database system.
- Ø These requirements/features for the new database system are described in documents known as the **requirements specifications**.
- Ø Many techniques for gathering this information (fact-finding techniques)

### Database Design

**Database Design:** Creating a design for a database that will support the mission statement and mission objectives.

- Ø Data Modeling is in the Database Design Phase.
- Ø Building data model requires answering questions about entities, relationships, and attributes.

Three phases of database design:

- Conceptual database design
- Logical database design
- Physical database design.

### Conceptual Database Design

Process of constructing a model of the data used, independent of all physical considerations.

- Ø Conceptual data model is built using the information in users' requirements specification.

Ø Ex. ER Diagram

### Logical Database Design

Conceptual data model is independent of all physical considerations, a logical model is derived knowing the underlying data model of the target DBMS.

Ø Ex. relational data model, normalization

### Physical Database Design

The physical design of the database specifies a description of the physical configuration of the database, such as the tables, file organizations, indexes, security, data types, and other parameters in the data dictionary.

Ø To describe how we intend to physically implement the logical database design.

### DBMS Selection

Selection of an appropriate DBMS to support the database system (if none exist).

Ø Undertaken at any time prior to logical design provided sufficient information is available regarding system requirements.

Ø Check off DBMS features against requirements.

Ø Some DBMS examples include MySQL, Microsoft Access, SQL Server, Oracle

### Example DBMS Evaluation Features

Data definition	Physical definition
Primary key enforcement	File structures available
Foreign key specification	File structure maintenance
Data types available	Ease of reorganization
Data type extensibility	Indexing
Domain specification	Variable length fields/records
Ease of restructuring	Data compression
Integrity controls	Encryption routines
View mechanism	Memory requirements
Data dictionary	Storage requirements
Data independence	
Underlying data model	
Schema evolution	
Accessibility	Transaction handling
Query language: SQL2/SQL:2003/ODMG compliant	Backup and recovery routines
Interfacing to 3GLs	Checkpointing facility
Multi-user	Logging facility
Security	Granularity of concurrency
– Office Access controls	Deadlock resolution strategy
– Authorization mechanism	Advanced transaction models
	Parallel query processing

## **Application Design**

Design of user interface and application programs that use and process the database.

Ø Database design and application design are parallel activities.

## **Prototyping (Optional)**

Building working model of a database system.

Ø Does not contain all the required features.

Ø Purpose

- to identify features of a system that are inadequate;
- to suggest improvements or even new features;
- to clarify the users' requirements;
- to evaluate feasibility of a particular system design.

## **Implementation**

Physical realization of the database and application designs.

- Use DDL to create database schemas and empty database files
- Use DDL to create any specified user views.

## **Data Conversion and Loading**

Transferring any existing data into new database and converting any existing applications to run on new database.

Ø Only required when new database system is replacing an old system.

- DBMS normally has utility that loads existing files into new database.

Ø May be possible to convert and use application programs from old system for use by new system.

## **Testing**

Process of running the database system with the intent of finding errors.

Ø Use carefully planned test strategies and realistic data.

Ø Demonstrates that database and application programs appear to be working according to requirements.

## **Operational Maintenance**

Process of monitoring and maintaining database system following installation.

Ø Monitoring performance of system.

- if performance falls, may require tuning or reorganization of the database.

Ø Maintaining and upgrading database system (when required).

Incorporating new requirements into database application.

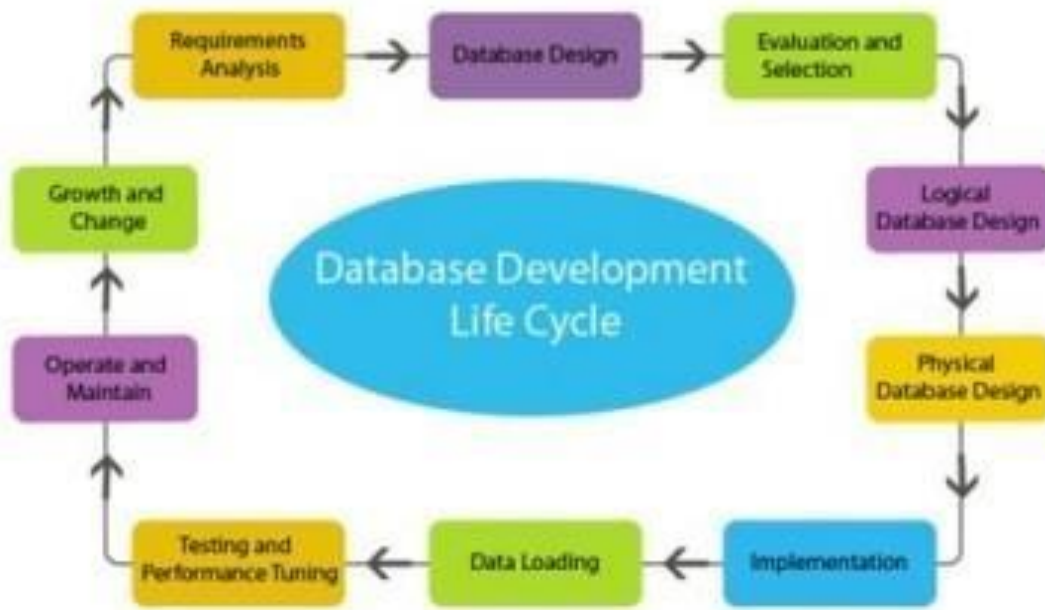
## **Explanation: 2**

The different phases of database development life cycle (DDLC) in the Database Management System (DBMS) are explained below –

- Requirement analysis.
- Database design.
- Evaluation and selection.
- Logical database design.



- Physical database design.
- Implementation.
- Data loading.
- Testing and performance tuning.
- Operation.
- Maintenance.



Now, let us understand these phases one by one.

### Requirement Analysis

The most important step in implementing a database system is to find out what is needed i.e what type of a database is required for the business organization, daily volume of data, how much data needs to be stored in the master files etc.

In order to collect all this information, a database analyst spends a lot of time within the business organization talking to people, end users and getting acquainted with the day-to-day process.

### Database Design

In this phase the database designers will make a decision on the database model that perfectly suits the organization's requirement. The database designers will study the documents prepared by the analysis in the requirement analysis stage and then start development of a system model that fulfils the needs.

### Evaluation and selection

In this phase, we evaluate the diverse database management systems and choose the one which perfectly suits the requirements of the organization.

In order to identify the best performing database, end users should be involved.

### Logical database design

Once the evaluation and selection phase is completed successfully, the next step is logical database design.

This design is translated into internal model which includes mapping of all objects i.e design of tables, indexes, views, transaction, access privileges etc.,

### **Physical Database Design**

This phase selects and characterizes the data storage and data access of the database. The data storage depends on the type of devices supported by the hardware, the data access methods. Physical design is very vital because of bad design which results in poor performance.

### **Implementation**

Database implementation needs the formation of special storage related constructs. These constructs consist of storage groups, table spaces, data files, tables etc.

### **Data Loading**

Once the database has been created, the data must be loaded into the database. The data required to be converted, if the loaded date is in a different format.

### **Operations**

In this phase, the database is accessed by the end users and application programs. This stage includes adding of new data, modifying existing data and deletion of absolute data. This phase provides useful information and helps management to make a business decision.

### **Maintenance**

It is one of the ongoing phases in DDLC.

The major tasks included are database backup and recovery, access management, hardware maintenance etc.

## **3.REQUIREMENT COLLECTION**

Before we can effectively design a database, we must know and analyze the expectations of the users and the intended uses of the database in as much detail as possible. This process is called **requirements collection and analysis**. To specify the requirements, we first identify the other parts of the information system that will interact with the database system. These include new and existing users and applications, whose requirements are then collected and analyzed. Typically, the following activities are part of this phase:

The major application areas and user groups that will use the database or whose work will be affected by it are identified. Key individuals and commit-tees within each group are chosen to carry out subsequent steps of requirements collection and specification.

Existing documentation concerning the applications is studied and analyzed. Other documentation—policy manuals, forms, reports, and organization charts—is reviewed to determine whether it has any influence on the requirements collection and specification process.

The current operating environment and planned use of the information is studied. This includes analysis of the types of transactions and their frequencies as well as of the flow of

information within the system. Geographic characteristics regarding users, origin of transactions, destination of reports, and so on are studied. The input and output data for the transactions are specified.

Written responses to sets of questions are sometimes collected from the potential database users or user groups. These questions involve the users' priorities and the importance they place on various applications. Key individuals may be interviewed to help in assessing the worth of information and in setting up priorities.

Requirement analysis is carried out for the final users, or *customers*, of the database system by a team of system analysts or requirement experts. The initial requirements are likely to be informal, incomplete, inconsistent, and partially incorrect. Therefore, much work needs to be done to transform these early requirements into a specification of the application that can be used by developers and testers as the starting point for writing the implementation and test cases. Because the requirements reflect the initial understanding of a system that does not yet exist, they will inevitably change. Therefore, it is important to use techniques that help customers converge quickly on the implementation requirements.

There is evidence that customer participation in the development process increases customer satisfaction with the delivered system. For this reason, many practitioners use meetings and workshops involving all stakeholders. One such methodology of refining initial system requirements is called Joint Application Design (JAD). More recently, techniques have been developed, such as Contextual Design, which involve the designers becoming immersed in the workplace in which the application is to be used. To help customer representatives better understand the proposed system, it is common to walk through workflow or transaction scenarios or to create a mock-up rapid prototype of the application.

The preceding modes help structure and refine requirements but leave them still in an informal state. To transform requirements into a better-structured representation, **requirements specification techniques** are used. These include object oriented analysis (OOA), data flow diagrams (DFDs), and the refinement of application goals. These methods use diagramming techniques for organizing and presenting information-processing requirements. Additional documentation in the form of text, tables, charts, and decision requirements usually accompanies the diagrams. There are techniques that produce a formal specification that can be checked mathematically for consistency and *what-if* symbolic analyses. These methods may become standard in the future for those parts of information systems that serve mission-critical functions and which therefore must work as planned. The model-based formal specification methods, of which the Z-notation and methodology is a prominent example, can be thought of as extensions of the ER model and are therefore the most applicable to information system design.

Some computer-aided techniques—called *Upper CASE* tools—have been proposed to help check the consistency and completeness of specifications, which are usually stored in a single repository and can be displayed and updated as the design progresses. Other tools are used to trace the links between requirements and other design entities, such as code modules and test cases. Such *traceability databases* are especially important in conjunction with enforced change-management procedures for systems where the requirements change frequently. They are also used in contractual projects where the development organization must provide documentary evidence to the customer that all the requirements have been implemented.

The requirements collection and analysis phase can be quite time-consuming, but it is crucial to the success of the information system. Correcting a requirements error is more expensive than correcting an error made during implementation because the effects of a requirements error are usually pervasive, and much more down-stream work has to be reimplemented as a result. Not correcting a significant error means that the system will not satisfy the customer and may not even be used at all. Requirements gathering and analysis is the subject of entire books.

### **When are Fact-Finding Techniques Used?**

Many situations arise for fact-finding during the database system development life cycle. However, fact-finding is particularly vital to the early stages of the life cycle, which includes database planning, system definition, and requirements gathering, and analysis stages. It is during these early stages where the database developer captures the necessary facts essential to build the required database. Fact-finding is also used in the case of database design and the later stages of the lifecycle but to a lesser extent. It is to be noted that it is important to make a rough estimation of how much time and effort is required to be spent on fact-finding for a database project.

### **What Facts Collect?**

Throughout the database system development life cycle, the database developer needs to confine the facts about the current and future systems. It is also true for the data captured and the citations produced at each stage. For example, problems come across during database design may necessitate additional data capture on the requirements for the new system.

### **Fact-Finding Techniques**

A database developer commonly uses several fact-finding techniques during a single database project. There are five widely used fact-finding techniques:

- Examining documentation
- Interviewing
- Observing the enterprise in action
- Research
- Questionnaires

Let us discuss in brief each of them:

1. **Examining documentation** can be helpful when you try to gain some insight as to how the requirement for a database arose. You may also find that documentation can help to acquire information on the part of the enterprise associated with the problem. If the problem relates to the current system, there should have to be documents associated with that system. By examining

documents, forms, reports, and files associated with the current system, you can quickly gain some thoughtful concepts out of the system.

2. **Interviewing** is the most frequently used, and usually the most useful, fact-finding procedure used. We can interview to collect information from person face-to-face. There can be several objectives for using interviewing, such as finding out facts, verifying those facts, clarifying these released facts, generating enthusiasm, getting the end-user involved, identifying requirements, and gathering ideas and opinions. However, using the interviewing practice must require proper communication skills for dealing effectively with people who have different values, priorities, opinions, motivations, and personalities.
3. **Observing the enterprise in action:** Observing the enterprise in action: Observation is one of the most successful fact-finding techniques carried out for understanding a system. Using this technique, it is achievable to either participate in or observe a person perform activities to learn about the system.
4. **Research:** A useful fact-finding technique is to research the application or the problem that you are dealing with and want to put within a database. Computer trade journals, reference books, and the Internet are good sources of information that can make available the vast quantity of information on how others have solved similar problems/issues plus whether or not any software packages exist to resolve or even partially solve your current problem.
5. **Questionnaires:** Another fabulous fact-finding method is to conduct surveys through questionnaires. Questionnaires are special-purpose documents that allow facts to be gathered from a large number of people while upholding some control over their responses. When dealing with a large number of listeners or audience, no other fact-finding technique can tabulate the same facts so efficiently. There are two types of questions that can be asked in a questionnaire, namely free-format and fixed-format. Free-format questions offer the respondent greater freedom inputting answers. Fixed-format questions require specific responses from individuals, and for the given question, the respondent must choose from the available answers.

## Collecting Data

Collecting data is relatively easy, but turning raw information into something useful requires that you know how to extract precisely what you need. In this module, intermediate to experienced programmers interested in data analysis will learn techniques for working with data in a business environment. You will learn how to look at data to discover what it contains, how to capture those ideas in conceptual models, and then feed your understanding back into the organization through business plans, metrics dashboards, and other applications. Along the way, you will experiment with concepts through hands-on exercises at various points in the module.

1. Use graphics to describe data with one, two, or dozens of variables
2. Develop conceptual models using back-of-the-envelope calculations, as well as scaling and probability arguments
3. Mine data with computationally intensive methods such as simulation and clustering
4. Make your conclusions understandable through reports, dashboards, and other metrics programs
5. Understand financial calculations, including the time value of money
6. Use dimensionality reduction techniques or predictive analytics to conquer challenging data analysis situations
7. Become familiar with different open source programming environments for data analysis

## DBLC Requirements Analysis

As mentioned earlier in this course, Requirements Analysis is the most important and most labor-intensive stage in the DBLC. It is critical for the designer to approach Requirements

Analysis armed with a plan for each **task** in the process.

Experience is the great teacher when it comes to assessing informational needs, but there is no substitute for preparation, specially for new designers. Most database designers begin Requirements Analysis by examining the existing database(s) to establish a framework for the remaining tasks. Analyzing how an organization stores data about its *business objects*<sup>[1]</sup>, and scrutinizing its perception of how it uses stored data (for example, gaining familiarity with its *business rules*)<sup>[2]</sup> provides that framework.

## Requirements Analysis

The goals of the requirements analysis are:

1. To determine the data requirements of the database in terms of primitive objects
2. To classify and describe the information about these objects
3. To identify and classify the relationships among the objects
4. To determine the types of transactions that will be executed on the database and the interactions between the data and the transactions
5. To identify rules governing the integrity of the data

The modeler works with the end users of an organization to determine the data requirements of the database. Information needed for the requirements analysis can be gathered in several ways:

**Review of existing documents:** Such documents include existing forms and reports, written guidelines, job descriptions, and personal narratives. Paper documentation is a good way to become familiar with the organization or activity you need to model.

### Tips for Successful Requirements Collection

Following are some of the tips for making the requirements collection process successful:

- Never assume that you know the customer's requirements. What you usually think, could be quite different to what the customer wants. Therefore, always verify with the customer when you have an assumption or a doubt.
- Get the end-users involved from the start. Get their support for what you do. • At the initial levels, define the scope and get customer's agreement. This helps you to successfully focus on scope of features.
- When you are in the process of collecting the requirements, make sure that the requirements are realistic, specific and measurable.
- Focus on making the requirements document crystal clear. Requirement document is the only way to get the client and the service provider to an agreement. Therefore, there should not be any gray area in this document. If there are gray areas, consider this would lead to potential business issues.
- Do not talk about the solution or the technology to the client until all the requirements are gathered. You are not in a position to promise or indicate anything to the client until

you are clear about the requirements.

- Before moving into any other project phases, get the requirements document signed off by the client.
- If necessary, create a prototype to visually illustrate the requirements.

Finalizing the requirements of the system to be built forms the backbone for the ultimate success of the project. It not only includes ascertaining the functions, but also the constraints of the system. The later part is very important as the customer needs to be very clear about the services that are going to be offered by the system. This will avoid any conflicts during the delivery or intermediate meetings with the client as the client assumes that the system provides those functions which are actually constraints of the system.

When the requirements of the system are inaccurate, it may lead to the following problems:

1. Delivery schedules may be slipped.
2. Developed system may be rejected by the client leading to the loss of reputation and amount spent on the project.
3. System developed may be unreliable.
4. Overall cost of the project may exceed the estimates.

There are different ways of finding the system requirements. Two of them are joint application development and prototyping.

## **4. DATABASE DESIGN**

### **What is Database Design?**

**Database Design** is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems. Properly designed database are easy to maintain, improves data consistency and are cost effective in terms of disk storage space. The database designer decides how the data elements correlate and what data must be stored.

The main objectives of database design in DBMS are to produce logical and physical designs models of the proposed database system.

The logical model concentrates on the data requirements and the data to be stored independent of physical considerations. It does not concern itself with how the data will be stored or where it will be stored physically.

The physical data design model involves translating the logical DB design of the database onto physical media using hardware resources and software systems such as database management systems (DBMS).

### **In this Database design tutorial, you will learn-**

- Why Database Design is Important?
- Database development life cycle

- Requirements analysis
- Database designing
- Implementation
- Types of Database Techniques

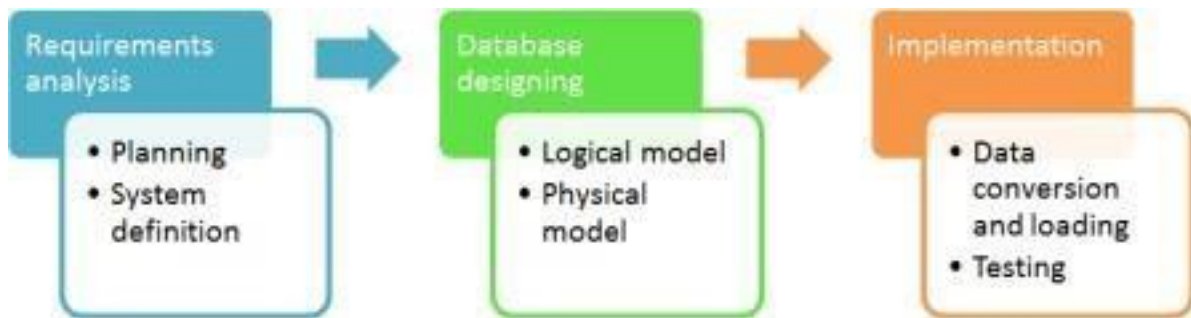
### Why Database Design is Important?

It helps produce database systems

1. That meet the requirements of the users
2. Have high performance.

Database design process in DBMS is crucial for **high performance** database system. Note, the genius of a database is in its design. Data operations using SQL is relatively simple

### Database development life cycle



### Requirements analysis

- **Planning** – This stages of database design concepts are concerned with planning of entire Database Development Life Cycle. It takes into consideration the Information Systems strategy of the organization.
- **System definition** – This stage defines the scope and boundaries of the proposed database system.

### Database designing

- **Logical model** – This stage is concerned with developing a database model based on requirements. The entire design is on paper without any physical implementations or specific DBMS considerations.
- **Physical model** – This stage implements the logical model of the database taking into account the DBMS and physical implementation factors.

### Implementation

- **Data conversion and loading** – this stage of relational databases design is concerned with importing and converting data from the old system into the new database.
- **Testing**
  - this stage is concerned with the identification of errors in the newly implemented system. It checks the database against requirement specifications.



The methodology is depicted as a bit by bit guide to the three main phases of database design, namely: **conceptual, logical, and physical design**.

The primary aim of each phase is as follows:

- **Conceptual database design** - to build the conceptual representation of the database, which has the identification of the important entities, relationships, and attributes.
- **Logical database design** - to convert the conceptual representation to the logical structure of the database, which includes designing the relations.
- **Physical database design** - to decide how the logical structure is to be physically implemented (as base relations) in the target Database Management System (DBMS).

## Database Design

Database design is the process of creating a design that will support the enterprise's mission statement and mission objectives for the required database system. Two main approaches to the design of a database are followed. These are:

- bottom-up and
- top-down

The **bottom-up approach** starts at the fundamental level of attributes (i.e., properties of entities and relationships), which through analysis of the associations between attributes, are clustered into relations that signify types of entities and relationships between entities.

A more appropriate strategy for the design of complex databases is to use the **top-down approach**, which starts with the development of data models that holds few high-level entities and relationships and then apply consecutive top-down refinements to identify lower-level entities, relationships, and the associated attributes. The top-down approach can be understood better using the concepts of the Entity-Relationship (ER) model, beginning with the identification of entities and relationships between the entities, which are of interest to the organization.

## Introduction to the Database Design Methodology

A structured approach that uses procedures, techniques, tools, and documentation help to support and make possible the process of design is called Design Methodology.

A design methodology encapsulates various phases, each containing some stages, which guide the designer in the techniques suitable at each stage of the project. A design methodology also helps the designer to plan, manage, control, and evaluate database development and managing projects. Furthermore, it is a planned approach for analyzing and modeling a group of requirements for a database in a standardized and ordered manner.

This stage is made up of three phases:

- Conceptual
- Logical and
- Physical database design

## **Conceptual Database Design**

In this design methodology, the process of constructing a model of the data is used in an enterprise, independent of all physical considerations. The conceptual database design phase starts with the formation of a conceptual data model of the enterprise that is entirely independent of implementation details such as the target DBMS, use of application programs, programming languages used, hardware platform, performance issues, or any other physical deliberations.

## **Critical Success Factors in Database Design**

The following planning strategies are often critical to the success of database design:

- Deal with task interactively with the users as much as possible.
- Follow a prearranged methodology throughout the data modeling process.
- Make use of a data-driven approach.
- Incorporate structural and integrity considerations into the data models.
- Combine conceptualization, normalization, and transaction validation methods into the data modeling methodology.
- Use figures for representing as much of the data models as possible.
- Use a Database Design Language (DBDL) to represent additional data semantics that cannot usually be represented in a diagram.
- Build a data dictionary to add-on the data model diagrams and the DBDL.
- Be willing to repeat steps.

These factors are constructed into the methodology that is presented for database design.

## **What are the steps for Conceptual Database Design?**

Conceptual database design steps are:

- Build a conceptual data model
- Recognize entity types
- Recognize the relationship types
- Identify and connect attributes with entity or relationship types
- Determine attribute domains
- Determine candidate, primary, and alternate key attributes
- Consider the use of improved modeling concepts (optional step)
- Check model for redundancy
- Validate the conceptual model against user transactions
- Review the conceptual data model with user

## **Building a Conceptual Data Model**

The first step in conceptual database design is to build one (or more) conceptual data replica of the data requirements of the enterprise. A conceptual data model comprises these following elements:

- entity types
- types of relationship

- attributes and the various attribute domains
- primary keys and alternate keys
- integrity constraints

The conceptual data model is maintained by documentation, including ER diagrams and a data dictionary, which is produced throughout the development of the model.

you have already come across the basics of what methodologies are and their stages. You have gathered the basic concept of what conceptual methodology is and how it works within the main stages of the database system development life cycle.

### **Details on Logical Methodology**

A local logical data model is used to characterize the data requirements of one or more but not all user views of a database, and a universal logical data model represents the data requirements for all user views. The final step of the logical database design phase is to reflect on how well the model can support possible future developments for the database system.

### **Logical Database Design Methodology for the Relational Model**

The objective of logical database design methodology is to interpret the conceptual data model into a logical data model and then authorize this model to check whether it is structurally correct and able to support the required transactions or not.

In this step of the database development life cycle, the main purpose is to translate the conceptual data model created in conceptual methodology (of the previous chapter) into a logical data model of the data requirements of the enterprise. This objective can be achieved by following the activities given below:

- Obtain the relations for the logical data model
- Authorize those relations using normalization
- Validate those relations against user transactions
- Check integrity control and its limitation
- Evaluate the logical data model with user
- Combine logical data models into the global model (This step is an optional one)
- Check for future growth and development

The structure of the relational schema is authorized using normalization. It then makes sure to ensure that the relations are capable of supporting the transactions given in the users' requirements specification. You can then check those all-important integrity constraints that are characterized by the logical data model. At this stage, the logical data model is authorized by the users to ensure that they consider the model to be a true demonstration of the data requirements for the enterprise.

### **Derive Relations for Logical Data Model**

The relationship that an entity has with other entities is characterized using the primary key or foreign key's concept. In deciding where to post the foreign key attribute(s), firstly, you must have to identify the 'parent' and 'child' entities that are involved in that relationship. The parent entity refers to the entity that posts a copy of its primary key into the relation that represents the child entity to act as the foreign key. You can describe how relations are obtained for the following

structures that may occur in a conceptual data model:

- strong entity types
- weak entity types
- one-to-many (1:\*) binary relationship types
- one-to-one (1:1) binary relationship types
- one-to-one (1:1) recursive relationship types
- superclass/subclass relationship types
- many-to-many (\*:\*) binary relationship types
- complex relationship types
- multi-valued attributes

### **Validate Relations Using Normalization**

In the previous step, you have derived a set of relations from signifying the conceptual data model created in the earlier step. Now, in the next step, you have to validate the groupings of attributes in each relation using the rules of normalization. The purpose of normalization is to ensure that the position of relations has a minimal and yet sufficient number of attributes necessary to support the data requirements of the enterprise.

### **Validate Relations against User Transactions**

The primary purpose of this step is to validate the logical data model to make certain that the model supports the required transactions, as the users' requirements specification. By using the relations, the primary key / foreign key links within the relations, the ER diagram, and the data dictionary, you can attempt to perform the operations manually. If you can resolve all transactions in this way, you can validate the logical data model against the transactions.

This physical methodology is the third and final phase of the database design methodology. Here, the designer must decide how to translate the logical database design (i.e., the entities, attributes, relationships, and constraints) into a physical database design, which can ultimately be implemented using the target DBMS. As the various parts of physical database design are highly reliant on the target DBMS, there may be more than one method of implementing any given portion of the database. Consequently, to do this work appropriately, the designers must be fully aware of the functionality of the target DBMS. They must recognize the advantages and disadvantages of each alternative approach for a particular accomplishment. For some systems, the designer may also need to select a suitable storage space/strategy that can take account of intended database usage.

### **What is Physical Database Design?**

It is the process of making a description of the execution of the database on secondary storage, which describes the base relations, file organizations as well as indexes used to gain efficient access to the data and any associated integrity constraints and security measures.

### **Comparison of Logical and Physical Database Design**

In designing and presenting a database design methodology, you have to divide the design process into three main stages or steps, also known as the Database development life cycle. These steps or stages are:

- Conceptual
- Logical and
- Physical database design

The phase before the physical design is the logical database design, which is largely independent of implementation details, such as the specific functionality of the target DBMS and application programs, but is reliant on the target data model. The outcome of this process is a logical data model that consists of an ER/relation diagram, relational schema, and supporting documents that depict this model, such as a data dictionary.

Logical database designs are concerned with the "what," and in contrast, physical database design is concerned with the "how." It requires diverse skills that are often found in different people. In particular, the physical database designer must know how the computer system hosts the DBMS and how it operates and must be fully conscious of the working of the target DBMS.

### **Steps Required for Implementing Physical Methodology**

The steps of the physical database design methodology are as follows:

- Transform the logical data model for target DBMS
  - Design base relations
  - Design representation of derived data
  - Design general constraints
- Design file organizations and indexes
  - Analyze transactions
  - Choose file organizations
  - Choose indexes
  - Estimate disk space requirements
  - Design user views
  - Design security mechanisms
  - Consider the introduction of controlled redundancy
  - Monitor and tune the operational system

### **Common Characteristics of a Physical Data Model**

- It typically illustrates data requirements for a single project or application. Sometimes even a part of an application
- May be incorporated into other physical data models by means of a repository of shared entities
- It typically includes 10-1000 tables; although these numbers are highly variable, depending on the scope of the data model
- It has the relationships between tables that address cardinality and nullability (optionality) of the relationships
- Designed and developed to be reliant on a specific version of a DBMS, storage location of data or technology
- Database columns will have data types with accurate precisions and lengths assigned to them. Columns will have nullability (optional) assigned
- Tables and columns will have specific definitions

## 5. What is ER Modeling?

An **Entity–relationship model (ER model)** describes the structure of a database with the help of a diagram, which is known as **Entity Relationship Diagram (ER Diagram)**. An ER model is a design or blueprint of a database that can later be implemented as a database. The main components of E-R model are: entity set and relationship set.

### What is an Entity Relationship Diagram (ER Diagram)?

An ER diagram shows the relationship among entity sets. An entity set is a group of similar entities and these entities can have attributes. In terms of DBMS, an entity is a table or attribute of a table in database, so by showing relationship among tables and their attributes, ER diagram shows the complete logical structure of a database. Lets have a look at a simple ER diagram to understand this concept.

### **ENTITY**

An **entity** is an object that exists and is distinguishable from other objects.

**Example: specific person, company, event, plant**

An **entity set** is a set of entities of the same type that share the same properties.

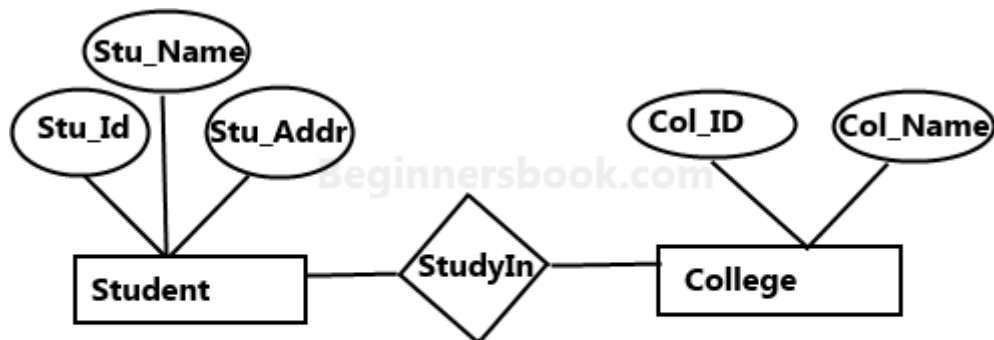
**Example: set of all persons, companies, trees, holidays**

### **E-R DIAGRAM (ENTITY RELATIONSHIP DIAGRAM)**

ER-Diagram is a visual representation of data that describes how data is related to each other.

- E-R Model was proposed by Dr. Peter Chen
- E-R model is graphical in nature, thus making it easy to analyze and Observe relationship between data elements
- Most DBMS are based upon E-R model

A simple ER Diagram:



**Sample E-R Diagram**

In the following diagram we have two entities Student and College and their relationship. The relationship between Student and College is many to one as a college can have many students however a student cannot study in multiple colleges at the same time. Student entity has attributes such as Stu\_Id, Stu\_Name & Stu\_Addr and College entity has attributes such as Col\_ID & Col\_Name.

Here are the geometric shapes and their meaning in an E-R Diagram. We will discuss these terms in detail in the next section(Components of a ER Diagram) of this guide so don't worry too much about these terms now, just go through them once.

**Rectangle:** Represents Entity sets.

**Ellipses:** Attributes

**Diamonds:** Relationship Set

**Lines:** They link attributes to Entity Sets and Entity sets to Relationship Set

**Double Ellipses:** Multivalued Attributes

**Dashed Ellipses:** Derived Attributes

**Double Rectangles:** Weak Entity Sets

**Double Lines:** Total participation of an entity in a relationship set



Represents Entity



Represents Attribute



Represents Relationship



Links Attribute(s) to entity set(s) or  
Entity set(s) to Relationship set(s)



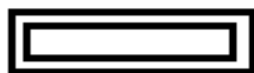
Represents Multivalued Attributes



Represents Derived Attributes



Represents Total Participation of Entity



Represents Weak Entity



Represents Weak Relationships



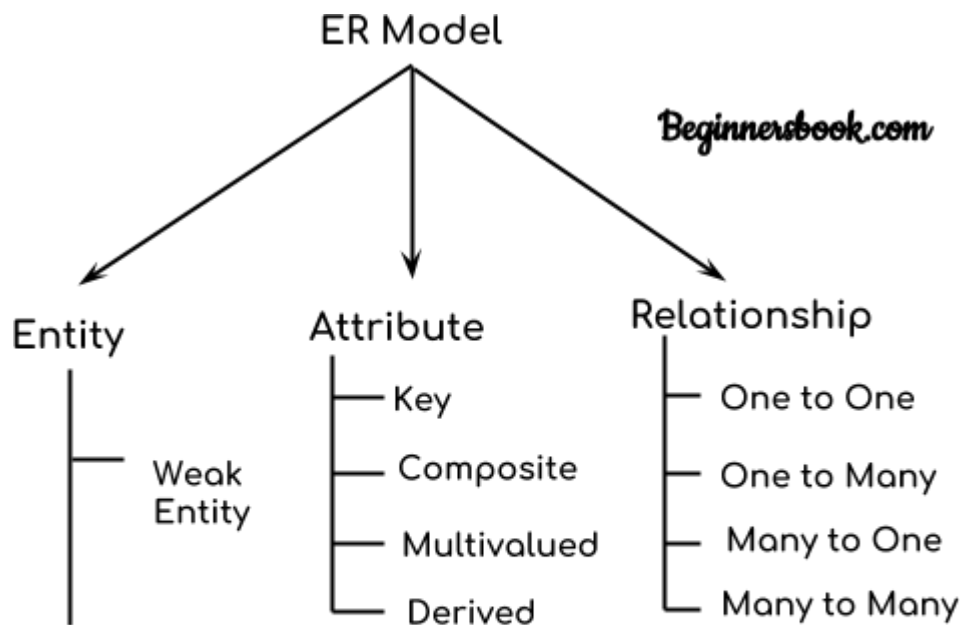
Represents Composite Attributes



Represents Key Attributes / Single Valued  
Attributes



## Components of a ER Diagram



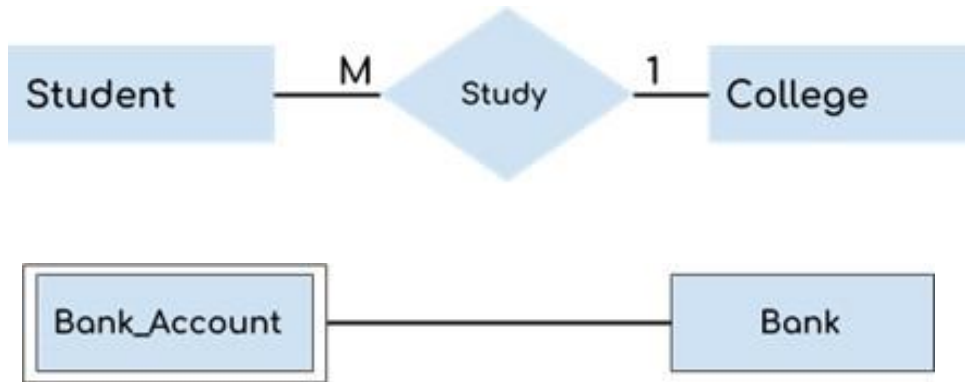
## Components of ER Diagram

As shown in the above diagram, an ER diagram has three main components:

1. Entity
2. Attribute
3. Relationship

### 1. Entity

An entity is an object or component of data. An entity is represented as rectangle in an ER diagram. For example: In the following ER diagram we have two entities Student and College and these two entities have many to one relationship as many students study in a single college. We will read more about relationships later, for now focus on entities.



**Weak Entity:**

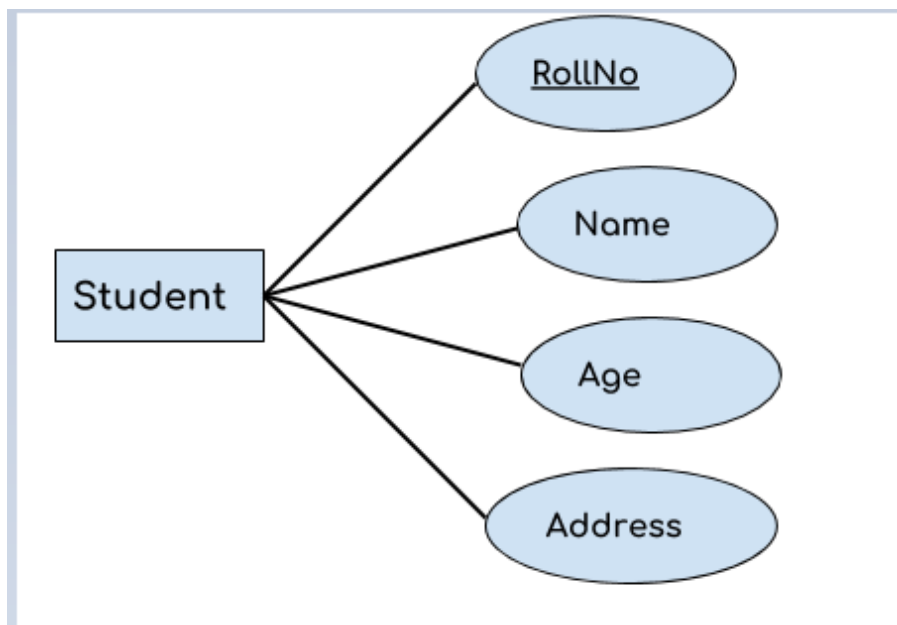
An entity that cannot be uniquely identified by its own attributes and relies on the relationship with other entity is called weak entity. The weak entity is represented by a double rectangle. For example – a bank account cannot be uniquely identified without knowing the bank to which the account belongs, so bank account is a weak entity.

**2. Attribute**

An attribute describes the property of an entity. An attribute is represented as Oval in an ER diagram. There are four types of attributes:

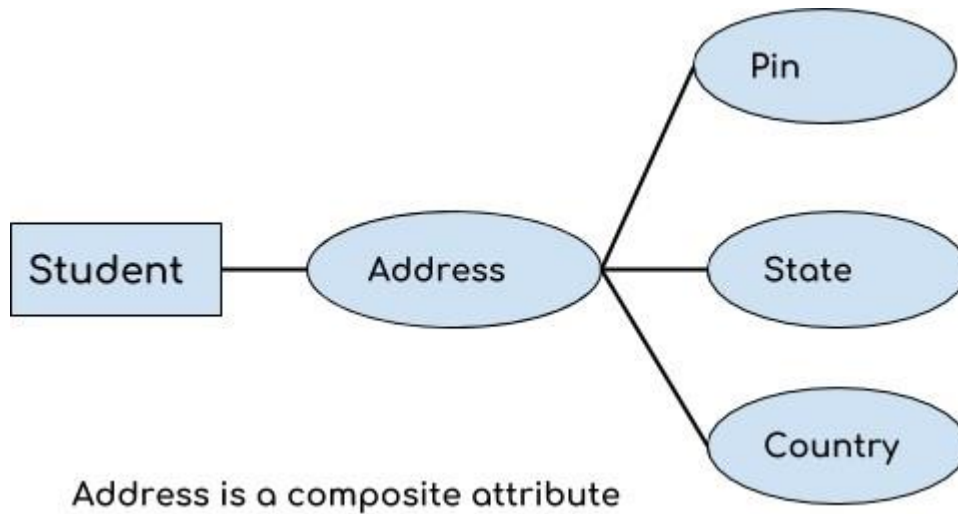
1. Key attribute
2. Composite attribute
3. Multivalued attribute
4. Derived attribute

**1. Key attribute:**



A key attribute can uniquely identify an entity from an entity set. For example, student roll number can uniquely identify a student from a set of students. Key attribute is represented by oval same as other attributes however the **text of key attribute is underlined**.

## 2. Composite attribute:



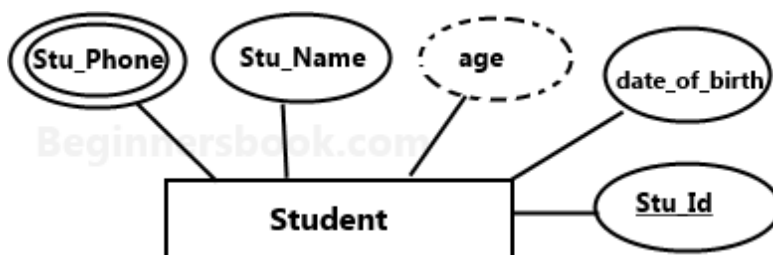
An attribute that is a combination of other attributes is known as composite attribute. For example, In student entity, the student address is a composite attribute as an address is composed of other attributes such as pin code, state, country.

## 3. Multivalued attribute:

An attribute that can hold multiple values is known as multivalued attribute. It is represented with **double ovals** in an ER Diagram. For example – A person can have more than one phone numbers so the phone number attribute is multivalued.

## 4. Derived attribute:

A derived attribute is one whose value is dynamic and derived from another attribute. It is represented by **dashed oval** in an ER Diagram. For example – Person age is a derived attribute as it changes over time and can be derived from another attribute (Date of birth).



## E-R diagram with multivalued and derived attributes:

### 3. Relationship

A relationship is represented by diamond shape in ER diagram, it shows the relationship among entities. There are four types of relationships:

1. One to One
2. One to Many
3. Many to One
4. Many to Many

#### 1. One to One Relationship



When a single instance of an entity is associated with a single instance of another entity then it is called one to one relationship. For example, a person has only one passport and a passport is given to one person.

#### 2. One to Many Relationship



When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationship. For example – a customer can place many orders but a order cannot be placed by many customers.

#### 3. Many to One Relationship



When more than one instances of an entity is associated with a single instance of another entity then it is called many to one relationship. For example – many students can study in a single college but a student cannot study in many colleges at the same time.

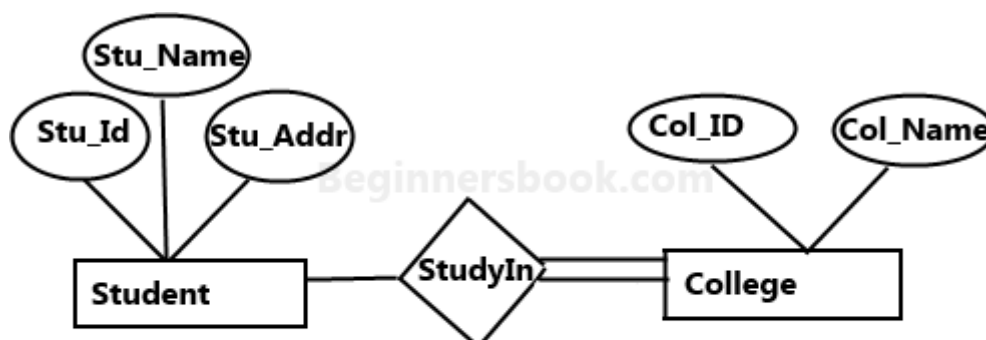
#### 4. Many to Many Relationship



When more than one instances of an entity is associated with more than one instances of another entity then it is called many to many relationship. For example, a can be assigned to many projects and a project can be assigned to many students.

#### Total Participation of an Entity set

Total participation of an entity set represents that each entity in entity set must have at least one relationship in a relationship set. It is also called **mandatory participation**. **For example:** In the following diagram each college must have at-least one associated Student. Total participation is represented using a **double line** between the entity set and relationship set.



**E-R Digram with total participation of College entity set in StudyIn relationship Set - This indicates that each college must have atleast one associated Student.**

#### Partial participation of an Entity Set

Partial participation of an entity set represents that each entity in the entity set may or may not participate in the relationship instance in that relationship set. It is also called as **optional participation**

Partial participation is represented using a single line between the entity set and relationship set.

**Example:** Consider an example of an IT company. There are many employees working for the company. Let's take the example of relationship between **employee** and role **software engineer**. Every software engineer is an employee but not every employee is software engineer as there are employees for other roles as well, such as housekeeping, managers, CEO etc. so we can say that participation of employee entity set to the software engineer relationship is partial.

## 6. Enhanced Entity Relationship (EER) Model

EER is a high-level data model that incorporates the extensions to the original ER model. Enhanced ERD are high level models that represent the requirements and complexities of complex database.

In addition to ER model concepts EE-R includes –

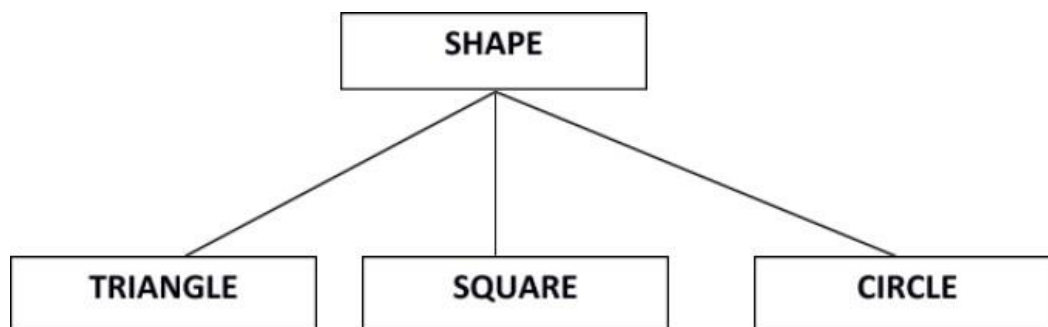
- Subclasses and Super classes.
- Specialization and Generalization.
- Category or union type.
- Aggregation.

These concepts are used to create EE-R diagrams.

Subclasses and Super class

Super class is an entity that can be divided into further subtype.

For **example** – consider Shape super class.

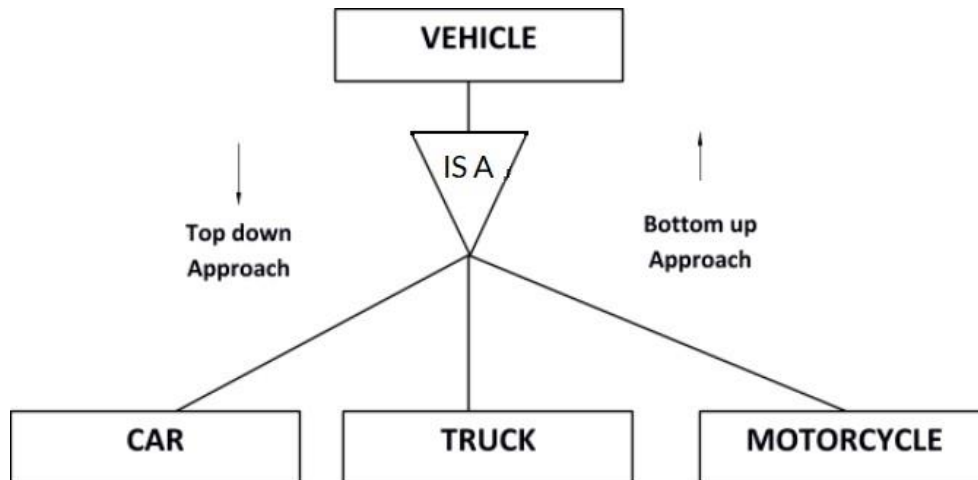


Super class shape has sub groups: Triangle, Square and Circle.

Sub classes are the group of entities with some unique attributes. Sub class inherits the properties and attributes from super class.

### Specialization and Generalization

Generalization is a process of generalizing an entity which contains generalized attributes or properties of generalized entities.



It is a Bottom up process i.e. consider we have 3 sub entities Car, Truck and Motorcycle. Now these three entities can be generalized into one super class named as Vehicle.

Specialization is a process of identifying subsets of an entity that share some different characteristic. It is a top down approach in which one entity is broken down into low level entity.

In above example Vehicle entity can be a Car, Truck or Motorcycle.

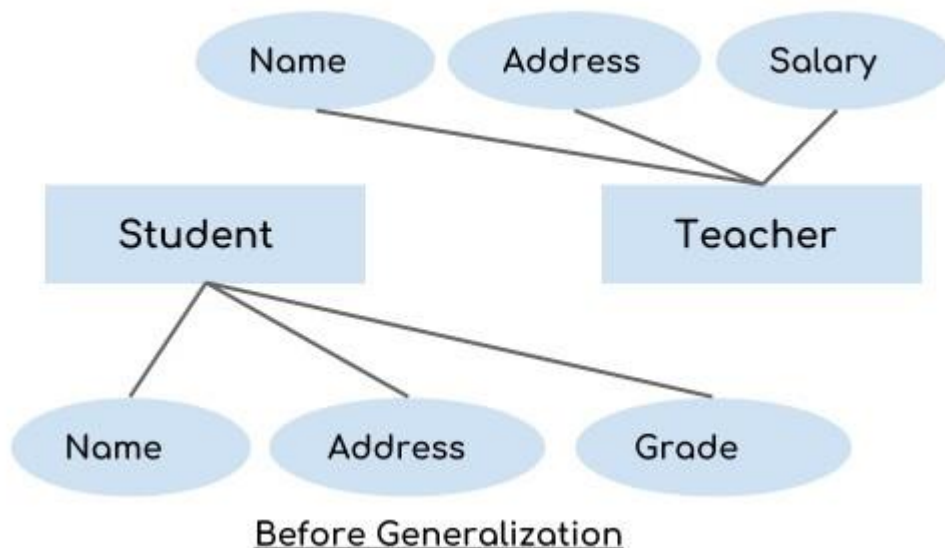
### Generalization Example

Lets say we have two entities Student and Teacher.

Attributes of Entity Student are: Name, Address & Grade

Attributes of Entity Teacher are: Name, Address & Salary

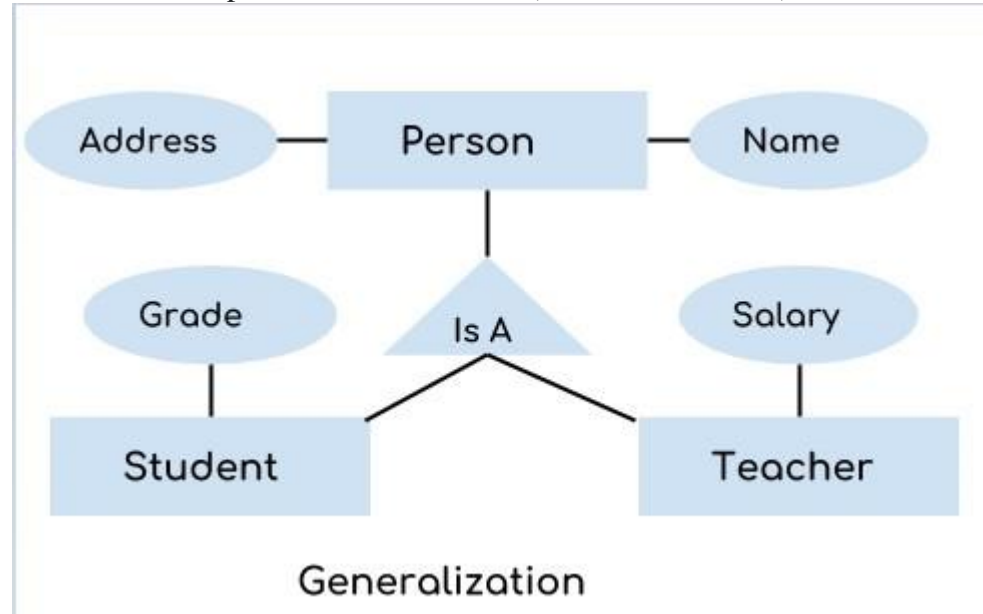
**The ER diagram before generalization looks like this:**



These two entities have two common attributes: Name and Address, we can make a generalized entity with these common attributes. Lets have a look at the ER model after generalization.

### The ER diagram after generalization:

We have created a new generalized entity Person and this entity has the common attributes of both the entities. As you can see in the following [ER diagram](#) that after the generalization process the entities Student and Teacher only has the specialized attributes Grade and Salary respectively and their common attributes (Name & Address) are now associated with a new entity Person which is in the relationship with both the entities (Student & Teacher).



### Note:

1. Generalization uses bottom-up approach where two or more lower level entities combine together to form a higher level new entity.
2. The new generalized entity can further combine together with lower level entity to create a further higher level generalized entity.

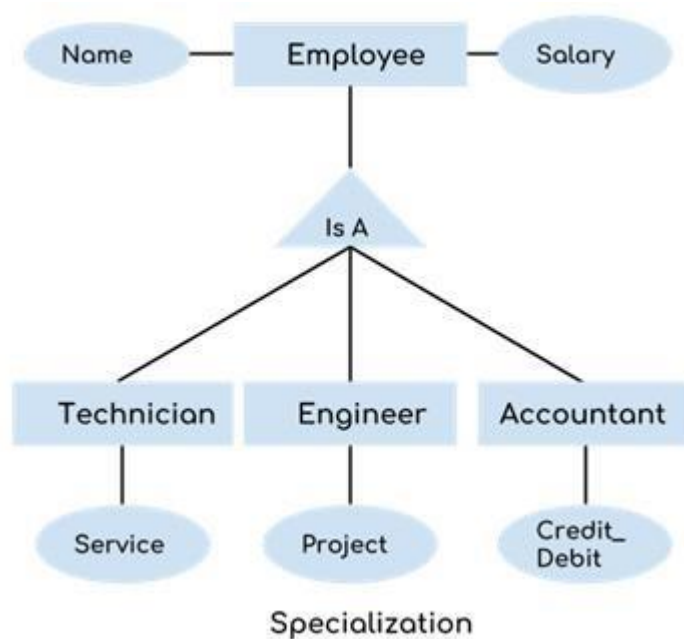
### DBMS Specialization

**Specialization** is a process in which an entity is divided into sub-entities. You can think of it as a reverse process of [generalization](#), in generalization two entities combine together to form a new higher level entity. Specialization is a top-down process.

The idea behind Specialization is to find the subsets of entities that have few distinguish attributes. For example – Consider an entity employee which can be further classified as sub-entities Technician, Engineer & Accountant because these sub entities have some distinguish attributes.



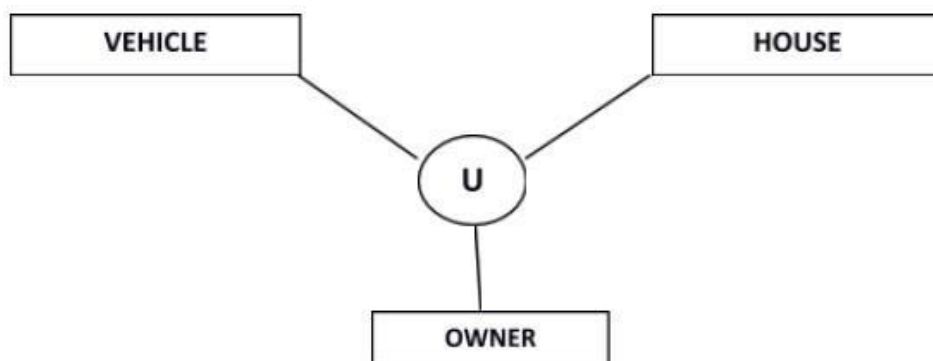
### Specialization Example



In the above diagram, we can see that we have a higher level entity –Employee which we have divided in sub entities –Technician, –Engineer & –Accountant. All of these are just an employee of a company, however their role is completely different and they have few different attributes. Just for the example, I have shown that Technician handles service requests, Engineer works on a project and Accountant handles the credit & debit details. All of these three employee types have few attributes common such as name & salary which we had left associated with the parent entity –Employee as shown in the above diagram.

### Category or Union

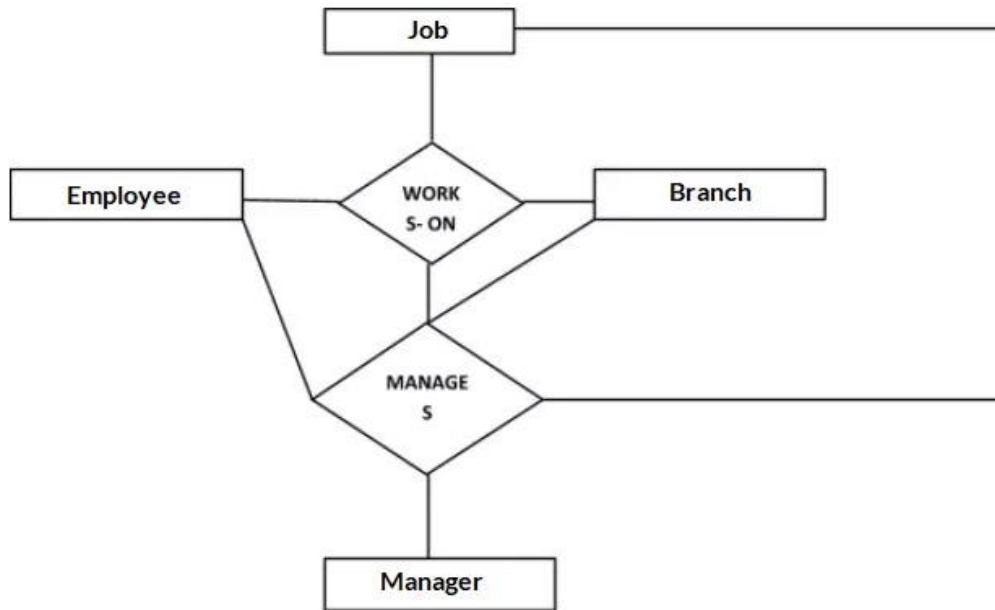
Relationship of one super or sub class with more than one super class.



Owner is the subset of two super class: Vehicle and House.

### Aggregation

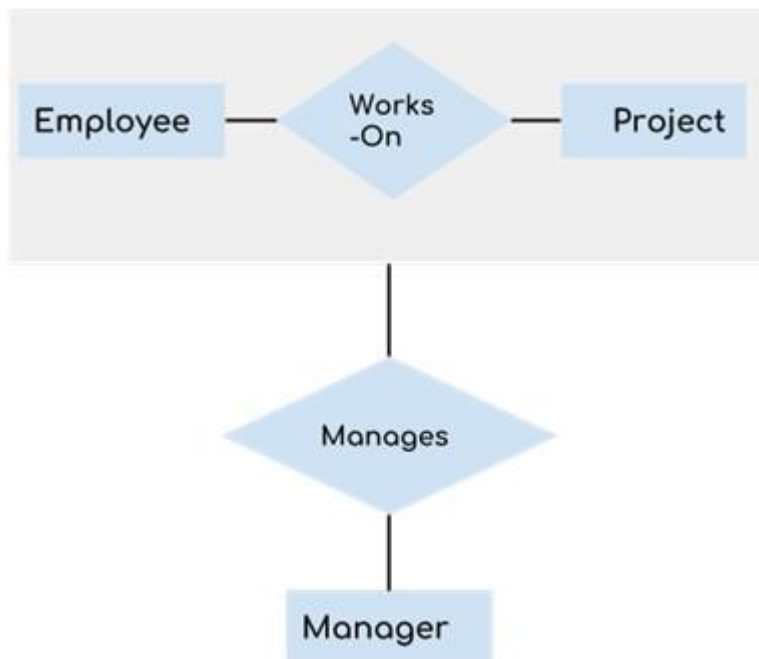
Represents relationship between a whole object and its component.



Consider a ternary relationship Works\_On between Employee, Branch and Manager. Now the best way to model this situation is to use aggregation, So, the relationship-set, Works\_On is a higher level entity-set. Such an entity-set is treated in the same manner as any other entity-set. We can create a binary relationship, Manager, between Works\_On and Manager to represent who manages what tasks.

**Aggregation** is a process in which a single entity alone is not able to make sense in a relationship so the relationship of two entities acts as one entity. I know it sounds confusing but don't worry the example we will take, will clear all the doubts.

### Aggregation Example



In real world, we know that a manager not only manages the employee working under them but he has to manage the project as well. In such scenario if entity –Manager‖ makes a –manages‖ relationship with either –Employee‖ or –Project‖ entity alone then it will not make any sense because he has to manage both. In these cases the relationship of two entities acts as one entity. In our example, the relationship –Works-On‖ between –Employee‖ & –Project‖ acts as one entity that has a relationship –Manages‖ with the entity –Manager‖.

### **Advantages of EER Models**

- It is quite simple to develop and maintain. In addition to this, it is easy to understand and interpret as well, technically speaking.
- Everything that is visually represented is easier to understand and maintain, and the same goes for EER models.
- It has been an efficient tool for database designers. It serves as a communication tool and helps display the relationship between entities.
- You can always convert the EER model into a table. Thus, it can easily be integrated into a relational model.

### **Disadvantages/Limitations of EER Diagrams**

- The EER diagrams have many constraints and come up with limited features.
- The Pareto Chart cannot be used for all the issues.
- Faults in the scoring of data can happen, plus also there could be an error in the application.
- Calculated on past data and therefore, cannot predict the future.

## **7.UML Class Diagram**

### **Unified Modeling Language (UML) | Class Diagrams**

#### **What is [UML](#)?**

It is the general-purpose modeling language used to visualize the system. It is a graphical language that is standard to the software industry for specifying, visualizing, constructing, and documenting the artifacts of the software systems, as well as for business modeling.

#### **Benefits of UML:**

- Simplifies complex software design, can also implement OOPs like a concept that is widely used.
- It reduces thousands of words of explanation in a few graphical diagrams that may reduce time consumption to understand.
- It makes communication more clear and more real.
- It helps to acquire the entire system in a view.
- It becomes very much easy for the software programmer to implement the actual demand once they have a clear picture of the problem.

**Types of UML:** The UML diagrams are divided into two parts: Structural UML diagrams and Behavioral UML diagrams which are listed below:

1. Structural UML diagrams
  - Class diagram
  - Package diagram
  - Object diagram
  - Component diagram
  - Composite structure diagram
  - Deployment diagram
2. Behavioral UML diagrams
  - Activity diagram
  - Sequence diagram
  - Use case diagram
  - State diagram
  - Communication diagram
  - Interaction overview diagram
  - Timing diagram

A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

It shows the attributes, classes, functions, and relationships to give an overview of the software system. It constitutes class names, attributes, and functions in a separate compartment that helps in software development. Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural diagram.

### Purpose of Class Diagrams

The main purpose of class diagrams is to build a static view of an application. It is the only diagram that is widely used for construction, and it can be mapped with object-oriented languages. It is one of the most popular UML diagrams. Following are the purpose of class diagrams given below:

1. It analyses and designs a static view of an application.
2. It describes the major responsibilities of a system.
3. It is a base for component and deployment diagrams.
4. It incorporates forward and reverse engineering.

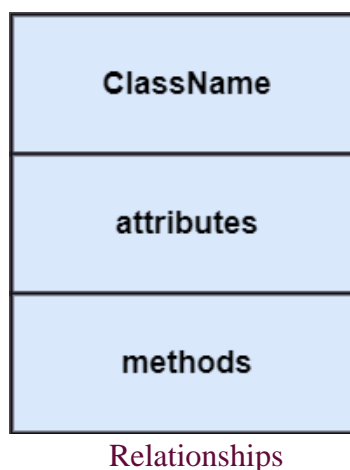
### Benefits of Class Diagrams

1. It can represent the object model for complex systems.
2. It reduces the maintenance time by providing an overview of how an application is structured before coding.
3. It provides a general schematic of an application for better understanding.
4. It represents a detailed chart by highlighting the desired code, which is to be programmed.
5. It is helpful for the stakeholders and the developers.

## Vital components of a Class Diagram

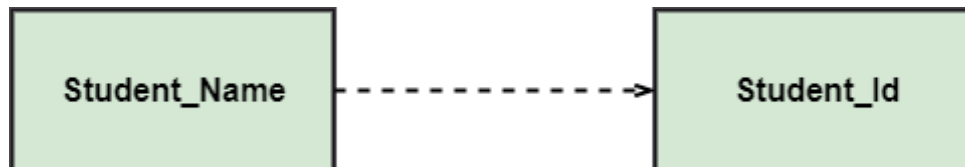
The class diagram is made up of three sections:

- **Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:
  - a. Capitalize the initial letter of the class name.
  - b. Place the class name in the center of the upper section.
  - c. A class name must be written in bold format.
  - d. The name of the abstract class should be written in italics format.
- **Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:
  - . The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).
    - a. The accessibility of an attribute class is illustrated by the visibility factors.
    - b. A meaningful name should be assigned to the attribute, which will explain its usage inside the class.
- **Lower Section:** The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.

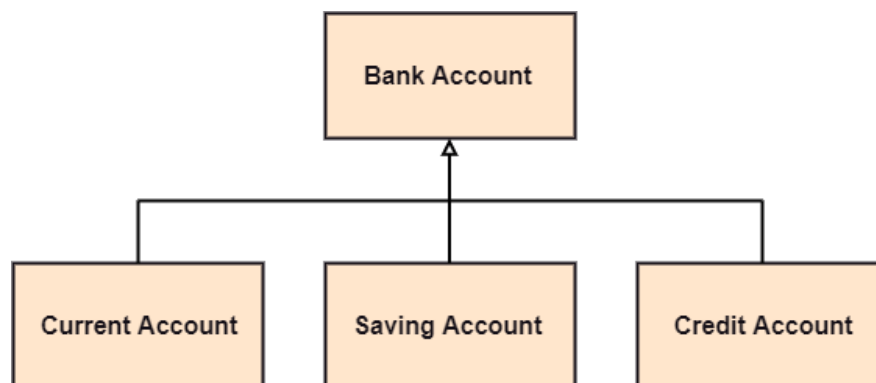


In UML, relationships are of three types:

- **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship. In the following example, Student\_Name is dependent on the Student\_Id.



- **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class. For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.

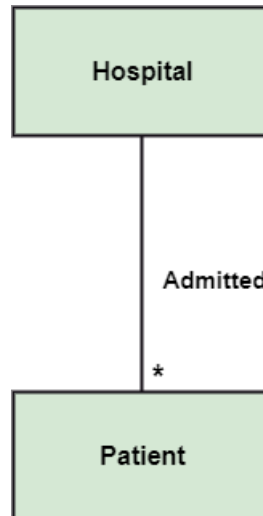


- **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship. For example, a department is associated with the college.



**Multiplicity:** It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.

For example, multiple patients are admitted to one hospital.



**Aggregation:** An aggregation is a subset of association, which represents has a relationship. It is more specific than association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.

The company encompasses a number of employees, and even if one employee resigns, the company still exists.



**Composition:** The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.

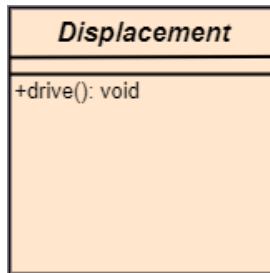
A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.



### Abstract Classes

In the abstract class, no objects can be a direct entity of the abstract class. The abstract class can neither be declared nor be instantiated. It is used to find the functionalities across the classes. The notation of the abstract class is similar to that of class; the only difference is that the name of the class is written in italics. Since it does not involve any implementation for a given function, it is best to use the abstract class with multiple objects.

Let us assume that we have an abstract class named **displacement** with a method declared inside it, and that method will be called as a **drive ()**. Now, this abstract class method can be implemented by any object, for example, car, bike, scooter, cycle, etc.



### How to draw a Class Diagram?

The class diagram is used most widely to construct software applications. It not only represents a static view of the system but also all the major aspects of an application. A collection of class diagrams as a whole represents a system.

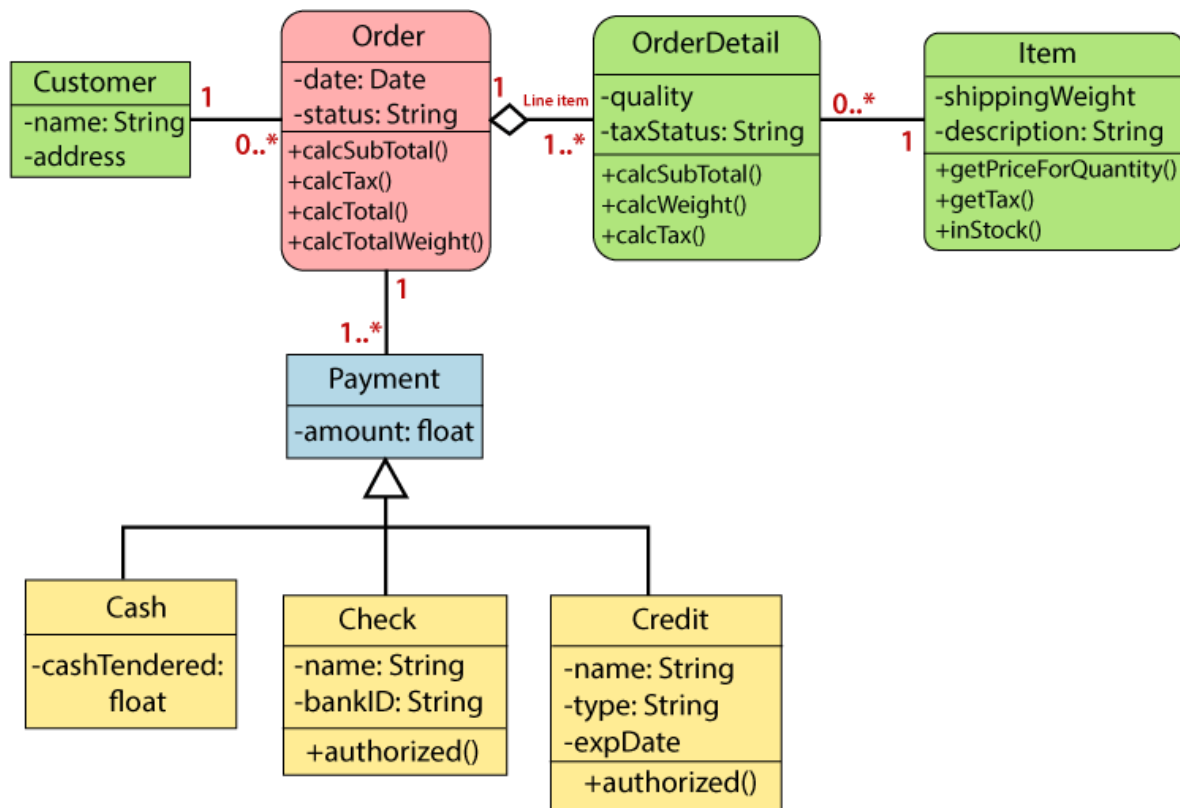
Some key points that are needed to keep in mind while drawing a class diagram are given below:

1. To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
2. The objects and their relationships should be acknowledged in advance.
3. The attributes and methods (responsibilities) of each class must be known.
4. A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
5. Notes can be used as and when required by the developer to describe the aspects of a diagram.
6. The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

### Class Diagram Example

A class diagram describing the sales order system is given below.





### Usage of Class diagrams

The class diagram is used to represent a static view of the system. It plays an essential role in the establishment of the component and deployment diagrams. It helps to construct an executable code to perform forward and backward engineering for any system, or we can say it is mainly used for construction. It represents the mapping with object-oriented languages that are C++, Java, etc. Class diagrams can be used for the following purposes:

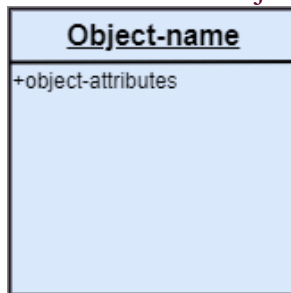
1. To describe the static view of a system.
2. To show the collaboration among every instance in the static view.
3. To describe the functionalities performed by the system.
4. To construct the software application using object-oriented languages.

### UML Object Diagram

Object diagrams are dependent on the class diagram as they are derived from the class diagram. It represents an instance of a class diagram. The objects help in portraying a static view of an object-oriented system at a specific instant.

Both the object and class diagram are similar to some extent; the only difference is that the class diagram provides an abstract view of a system. It helps in visualizing a particular functionality of a system.

### Notation of an Object Diagram



### Purpose of Object Diagram

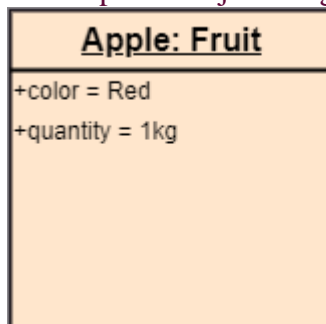
The object diagram holds the same purpose as that of a class diagram. The class diagram provides an abstract view which comprises of classes and their relationships, whereas the object diagram represents an instance at a particular point of time.

The object diagram is actually similar to the concrete (actual) system behavior. The main purpose is to depict a static view of a system.

Following are the purposes enlisted below:

- It is used to perform forward and reverse engineering.
- It is used to understand object behavior and their relationships practically.
- It is used to get a static view of a system.
- It is used to represent an instance of a system.

### Example of Object Diagram



### How to draw an Object Diagram?

1. All the objects present in the system should be examined before start drawing the object diagram.
2. Before creating the object diagram, the relation between the objects must be acknowledged.
3. The association relationship among the entities must be cleared already.
4. To represent the functionality of an object, a proper meaningful name should be assigned.
5. The objects are to be examined to understand its functionality.

### Applications of Object diagrams

The following are the application areas where the object diagrams can be used.

1. To build a prototype of a system.
2. To model complex data structures.
3. To perceive the system from a practical perspective.
4. Reverse engineering.

#### Class vs. Object diagram

Serial No.	Class Diagram	Object Diagram
1.	It depicts the static view of a system.	It portrays the real-time behavior of a system.
2.	Dynamic changes are not included in the class diagram.	Dynamic changes are captured in the object diagram.
3.	The data values and attributes of an instance are not involved here.	It incorporates data values and attributes of an entity.
4.	The object behavior is manipulated in the class diagram.	Objects are the instances of a class.

Relational model concepts -- Integrity constraints -- SQL Data manipulation  
– SQL Data definition – Views -- SQL programming.

## **WHAT IS RELATIONAL MODEL?**

**Relational Model (RM)** represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.

**Some popular Relational Database management systems are:**

- DB2 and Informix Dynamic Server – IBM
- Oracle and RDB – Oracle
- SQL Server and Access – Microsoft

In this tutorial, you will learn

- Relational Model Concepts in DBMS
- Relational Integrity Constraints
- Operations in Relational Model
- Best Practices for creating a Relational Model
- Advantages of Relational Database Model
- Disadvantages of Relational Model

## **Relational Model Concepts in DBMS**

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student\_Rollno, NAME,etc.
2. **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the relation with its attributes.
5. **Degree:** The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:** Total number of rows present in the Table.
7. **Column:** The column represents the set of values for a specific attribute.
8. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.

9. **Relation key** – Every row has one, two or multiple attributes, which is called relation key.
10. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain

### Table also called Relation

© guru99.com

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

**Primary Key** (points to CustomerID)

**Domain** (points to CustomerName, Ex: NOT NULL)

**Column OR Attributes** (points to the columns)

**Tuple OR Row** (points to the rows)

Total # of rows is **Cardinality**

Total # of column is **Degree**

## RELATIONAL INTEGRITY CONSTRAINTS

Relational Integrity constraints in DBMS are referred to conditions which must be present for a valid relation. These Relational constraints in DBMS are derived from the rules in the mini-world that the database represents.

There are many types of Integrity Constraints in DBMS. Constraints on the Relational database management system is mostly divided into three main categories are:

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

### Domain Constraints

Domain constraints can be violated if an attribute value is not appearing in the corresponding domain or it is not of the appropriate data type.

Domain constraints specify that within each tuple, and the value of each attribute must be unique. This is specified as data types which include standard data types integers, real numbers, characters, Booleans, variable length strings, etc.

### Example:

```
Create DOMAIN CustomerName
CHECK (value not NULL)
```

The example shown demonstrates creating a domain constraint such that CustomerName is not NULL

## Key Constraints

An attribute that can uniquely identify a tuple in a relation is called the key of the table. The value of the attribute for different tuples in the relation has to be unique.

### Example:

In the given table, CustomerID is a key attribute of Customer Table. It is most likely to have a single key for one customer, CustomerID =1 is only for the CustomerName =” Google”.

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

## Referential Integrity Constraints

Referential Integrity constraints in DBMS are based on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.

### Example:

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Customer

InvoiceNo	CustomerID	Amount
1	1	\$100
2	1	\$200
3	2	\$150

Billing

In the above example, we have 2 relations, Customer and Billing. Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount \$300

## Operations in Relational Model

Four basic update operations performed on relational database model are Insert, update, delete and select.


- Insert is used to insert data into the relation
- Delete is used to delete tuples from the table.
- Modify allows you to change the values of some attributes in existing tuples.
- Select allows you to choose a specific range of data.

Whenever one of these operations are applied, integrity constraints specified on the relational database schema must never be violated.

### Insert Operation

The insert operation gives values of the attribute for a new tuple which should be inserted into a relation.

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive



CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive
4	Alibaba	Active

### Update Operation

You can see that in the below-given relation table CustomerName= 'Apple' is updated from Inactive to Active.

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive
4	Alibaba	Active




CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Active
4	Alibaba	Active

### Delete Operation

To specify deletion, a condition on the attributes of the relation selects the tuple to be deleted.

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Active
4	Alibaba	Active



CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
4	Alibaba	Active

In the above-given example, CustomerName= "Apple" is deleted from the table. The Delete operation could violate referential integrity if the tuple which is deleted is referenced by foreign keys from other tuples in the same database.

### Select Operation

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
4	Alibaba	Active



CustomerID	CustomerName	Status
2	Amazon	Active

In the above-given example, CustomerName="Amazon" is selected

### **Best Practices for creating a Relational Model**

- Data need to be represented as a collection of relations
- Each relation should be depicted clearly in the table
- Rows should contain data about instances of an entity
- Columns must contain data about attributes of the entity
- Cells of the table should hold a single value
- Each column should be given a unique name
- No two rows can be identical
- The values of an attribute should be from the same domain

### **Advantages of Relational Database Model**

- **Simplicity:** A Relational data model in DBMS is simpler than the hierarchical and network model.
- **Structural Independence:** The relational database is only concerned with data and not with a structure. This can improve the performance of the model.
- **Easy to use:** The Relational model in DBMS is easy as tables consisting of rows and columns are quite natural and simple to understand
- **Query capability:** It makes possible for a high-level query language like SQL to avoid complex database navigation.
- **Data independence:** The Structure of Relational database can be changed without having to change any application.
- **Scalable:** Regarding a number of records, or rows, and the number of fields, a database should be enlarged to enhance its usability.

### **Disadvantages of Relational Model**

- Few relational databases have limits on field lengths which can't be exceeded.
- Relational databases can sometimes become complex as the amount of data grows, and the relations between pieces of data become more complicated.
- Complex relational database systems may lead to isolated databases where the information cannot be shared from one system to another.

### **Summary**

- The Relational database modelling represents the database as a collection of relations (tables)
- Attribute, Tables, Tuple, Relation Schema, Degree, Cardinality, Column, Relation instance, are some important components of Relational Model
- Relational Integrity constraints are referred to conditions which must be present for a valid Relation approach in DBMS
- Domain constraints can be violated if an attribute value is not appearing in the corresponding domain or it is not of the appropriate data type
- Insert, Select, Modify and Delete are the operations performed in Relational Model constraints



- The relational database is only concerned with data and not with a structure which can improve the performance of the model
- Advantages of Relational model in DBMS are simplicity, structural independence, ease of use, query capability, data independence, scalability, etc.
- Few relational databases have limits on field lengths which can't be exceeded.

## RELATIONAL MODEL CONCEPT

Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

**Domain:** It contains a set of atomic values that an attribute can take.

**Attribute:** It contains the name of a column in a particular table. Each attribute  $A_i$  must have a domain,  $dom(A_i)$

**Relational instance:** In the relational database system, the relational instance is represented by a finite set of tuples. Relation instances do not have duplicate tuples.

**Relational schema:** A relational schema contains the name of the relation and name of all columns or attributes.

**Relational key:** In the relational key, each row has one or more attributes. It can identify the row in the relation uniquely.

**Example: STUDENT Relation**

**NAME ROLL\_NO PHONE\_NO ADDRESS AGE**

Ram	14795	7305758992	Noida	24
Shyam	12839	9026288936	Delhi	35
Laxman	33289	8583287182	Gurugram	20
Mahesh	27857	7086819134	Ghaziabad	27
Ganesh	17282	9028 9i3988	Delhi	40

- In the given table, NAME, ROLL\_NO, PHONE\_NO, ADDRESS, and AGE are the attributes.
- The instance of schema STUDENT has 5 tuples.
- $t_3 = \langle \text{Laxman}, 33289, 8583287182, \text{Gurugram}, 20 \rangle$

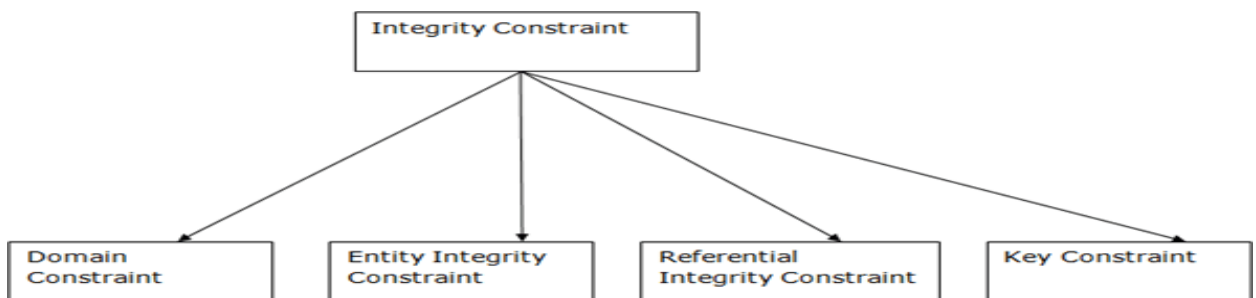
## Properties of Relations

- Name of the relation is distinct from all other relations.
- Each relation cell contains exactly one atomic (single) value
- Each attribute contains a distinct name
- Attribute domain has no significance
- tuple has no duplicate value
- Order of tuple can have a different sequence

## INTEGRITY CONSTRAINTS

- Integrity constraints are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.

### Types of Integrity Constraint



#### 1. Domain constraints

- Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

**Example:**

ID	NAME	SEMENSTER	AGE
1000	Tom	1 <sup>st</sup>	17
1001	Johnson	2 <sup>nd</sup>	24
1002	Leonardo	5 <sup>th</sup>	21
1003	Kate	3 <sup>rd</sup>	19
1004	Morgan	8 <sup>th</sup>	A

Not allowed. Because AGE is an integer attribute

#### 2. Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.

- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.

**Example:**

**EMPLOYEE**

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

### 3. Referential Integrity Constraints

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

**Example:**

(Table 1)

EMP_NAME	NAME	AGE	D_No
1	Jack	20	11
2	Harry	40	24
3	John	27	18
4	Devil	38	13

Foreign key

Not allowed as D\_No 18 is not defined as a Primary key of table 2 and In table 1, D\_No is a foreign key defined

Relationships

(Table 2)

<u>D_No</u>	D_Location
11	Mumbai
24	Delhi
13	Noida

Primary Key

### 4. Key constraints

- Keys are the entity set that is used to identify an entity within its entity set uniquely.

- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

**Example:**

ID	NAME	SEMENSTER	AGE
1000	Tom	1 <sup>st</sup>	17
1001	Johnson	2 <sup>nd</sup>	24
1002	Leonardo	5 <sup>th</sup>	21
1003	Kate	3 <sup>rd</sup>	19
1002	Morgan	8 <sup>th</sup>	22

Not allowed. Because all row must be unique

## Introduction to DDL

- DDL stands for **Data Definition Language**.
- It is a language used for defining and modifying the data and its structure.
- It is used to build and modify the structure of your tables and other objects in the database.

**DDL commands are as follows,**

1. CREATE
  2. DROP
  3. ALTER
  4. RENAME
  5. TRUNCATE
- These commands can be used to add, remove or modify tables within a database.
  - DDL has pre-defined syntax for describing the data.

### 1. CREATE COMMAND

- **CREATE command** is used for creating objects in the database.
- It creates a new table.

**Syntax:**

CREATE TABLE <table\_name>

( column\_name1 datatype,  
column\_name2 datatype,

.  
. .  
.

```
column_name_n datatype  
);
```

### **Example : CREATE command**

```
CREATE TABLE employee  
(  
    empid INT,  
    ename CHAR,  
    age INT,  
    city CHAR(25),  
    phone_no VARCHAR(20)  
);
```

## **2. DROP COMMAND**

- **DROP command** allows to remove entire database objects from the database.
- It removes entire data structure from the database.
- It deletes a table, index or view.

### **Syntax:**

```
DROP TABLE <table_name>;  
OR  
DROP DATABASE <database_name>;
```

### **Example : DROP Command**

```
DROP TABLE employee;  
OR  
DROP DATABASE employees;
```

- If you want to remove individual records, then use DELETE command of the DML statement.

## **3. ALTER COMMAND**

- An **ALTER command** allows to alter or modify the structure of the database.
- It modifies an existing database object.
- Using this command, you can add additional column, drop existing column and even change the data type of columns.

### **Syntax:**

```
ALTER TABLE <table_name>  
ADD <column_name datatype>;
```

OR

```
ALTER TABLE <table_name>
```

```
CHANGE <old_column_name> <new_column_name>;
```

OR

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

**Example : ALTER Command**

```
ALTER TABLE employee  
ADD (address varchar2(50));
```

OR

```
ALTER TABLE employee  
CHANGE (phone_no) (contact_no);
```

OR

```
ALTER TABLE employee  
DROP COLUMN age;
```

To view the changed structure of table, use 'DESCRIBE' command.

**For example:**

```
DESCRIBE TABLE employee;
```

#### **4. RENAME COMMAND**

- **RENAME command** is used to rename an object.
- It renames a database table.

**Syntax:**

```
RENAME TABLE <old_name> TO <new_name>;
```

**Example:**

```
RENAME TABLE emp TO employee;
```

#### **5. TRUNCATE COMMAND**

- **TRUNCATE command** is used to delete all the rows from the table permanently.
- It removes all the records from a table, including all spaces allocated for the records.
- This command is same as DELETE command, but TRUNCATE command does not generate any rollback data.

**Syntax:**

```
TRUNCATE TABLE <table_name>;
```

**Example:**

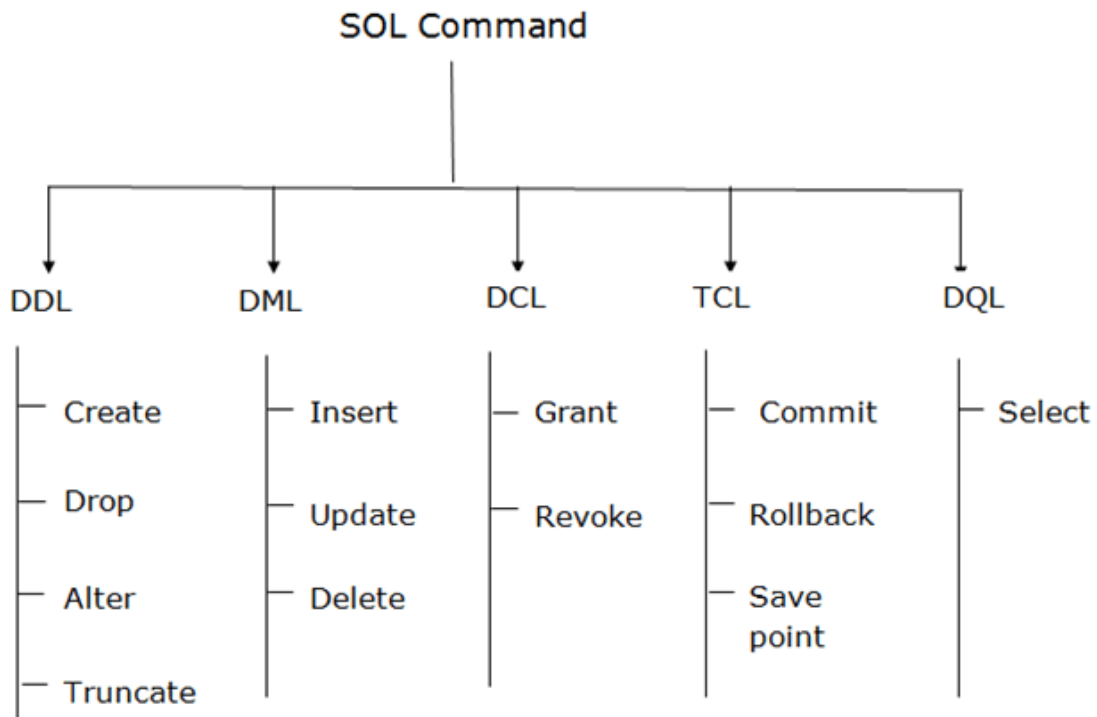
```
TRUNCATE TABLE employee;
```

# SQL COMMANDS

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

## Types of SQL Commands

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



### 1. Data Definition Language (DDL)

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

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

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

#### Syntax:

1. CREATE TABLE TABLE\_NAME (COLUMN\_NAME DATATYPES[. . .]);

#### Example:

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

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

**Syntax**

1. DROP TABLE table\_name;

**Example**

1. DROP TABLE EMPLOYEE;

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

**Syntax:**

To add a new column in the table

1. ALTER TABLE table\_name ADD column\_name COLUMN-definition;

To modify existing column in the table:

1. ALTER TABLE table\_name MODIFY(column\_definitions. ..);

**EXAMPLE**

1. ALTER TABLE STU\_DETAILS ADD(ADDRESS VARCHAR2(20));
2. ALTER TABLE STU\_DETAILS MODIFY (NAME VARCHAR2(20));

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

**Syntax:**

1. TRUNCATE TABLE table\_name;

**Example:**

1. TRUNCATE TABLE EMPLOYEE;

## 2. DATA MANIPULATION LANGUAGE

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

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

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

**Syntax:**

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

Or

1. INSERT INTO TABLE\_NAME
2. VALUES (value1, value2, value3, ..... valueN);

**For example:**

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



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

**Syntax:**

1. UPDATE table\_name SET [column\_name1= value1,...column\_nameN = valueN] [WHERE CONDITION]

**For example:**

1. UPDATE students
2. SET User\_Name = 'Sonoo'
3. WHERE Student\_Id = '3'

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

**Syntax:**

1. DELETE FROM table\_name [WHERE condition];

**For example:**

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

### 3. DATA CONTROL LANGUAGE

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

Here are some commands that come under DCL:

- Grant
- Revoke

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

**Example**

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

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

**Example**

1. REVOKE SELECT, UPDATE ON MY\_TABLE FROM USER1, USER2;

### 4. TRANSACTION CONTROL LANGUAGE

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

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

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

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

**Syntax:**

1. COMMIT;

**Example:**

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

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

**Syntax:**

1. ROLLBACK;

**Example:**

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

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

**Syntax:**

SAVEPOINT SAVEPOINT\_NAME;

## 5. DATA QUERY LANGUAGE

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

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

**Syntax:**

1. SELECT expressions
2. FROM TABLES
3. WHERE conditions;

**For example:**

1. SELECT emp\_name
2. FROM employee
3. WHERE age > 20;

## SQL VIEWS

### SQL CREATE VIEW Statement

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

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

A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
```

```
SELECT column1, column2, ...
```

```
FROM table_name
```

```
WHERE condition;
```

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

### SQL CREATE VIEW Examples

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

Example

```
CREATE VIEW [Brazil Customers] AS
```

```
SELECT CustomerName, ContactName
```

```
FROM Customers
```

```
WHERE Country = 'Brazil';
```

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

Example

```
CREATE VIEW [Products Above Average Price] AS
```

```
SELECT ProductName, Price
```

```
FROM Products
```

```
WHERE Price > (SELECT AVG(Price) FROM Products);
```

We can query the view above as follows:

Example

```
SELECT * FROM [Products Above Average Price];
```

### SQL Updating a View

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

SQL CREATE OR REPLACE VIEW Syntax

```
CREATE OR REPLACE VIEW view_name AS
```

```
SELECT column1, column2, ...
```

```
FROM table_name
```

```
WHERE condition;
```

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

Example

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

### SQL Dropping a View

A view is deleted with the DROP VIEW statement.

SQL DROP VIEW Syntax

```
DROP VIEW view_name;
```

The following SQL drops the "Brazil Customers" view:

Example

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

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following –

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

### Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows –

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

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records –

```
+--+-----+--+-----+-----+
```

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

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS;
```

Now, you can query CUSTOMERS\_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

name	age
Ramesh	32
Khilan	25
kaushik	23
Chaitali	25
Hardik	27
Komal	22
Muffy	24

### **The WITH CHECK OPTION**

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error. The following code block has an example of creating same view CUSTOMERS\_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS
WHERE age IS NOT NULL
```

WITH CHECK OPTION;

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below –

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
```

```
SET AGE = 35
```

```
WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+--+-----+--+-----+-----+
|ID|NAME  |AGE|ADDRESS |SALARY |
+--+-----+--+-----+-----+
| 1| Ramesh | 35| Ahmedabad | 2000.00 |
| 2| Khilan | 25| Delhi    | 1500.00 |
| 3| kaushik | 23| Kota     | 2000.00 |
| 4| Chaitali | 25| Mumbai  | 6500.00 |
| 5| Hardik | 27| Bhopal  | 8500.00 |
| 6| Komal  | 22| MP      | 4500.00 |
| 7| Muffy  | 24| Indore  | 10000.00 |
+--+-----+--+-----+-----+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS\_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

## Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
```

```
WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+--+-----+--+-----+-----+
|ID|NAME   |AGE|ADDRESS |SALARY |
+--+-----+--+-----+-----+
| 1|Ramesh  |35 |Ahmedabad| 2000.00|
| 2|Khilan  |25 |Delhi    | 1500.00|
| 3|kaushik |23 |Kota     | 2000.00|
| 4|Chaitali|25 |Mumbai   | 6500.00|
| 5|Hardik  |27 |Bhopal   | 8500.00|
| 7|Muffy   |24 |Indore   |10000.00|
+--+-----+--+-----+-----+
```

## Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below –

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS\_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

PL/pgSQL is PostgreSQL's built-in programming language for writing functions which run within the database itself, known as stored procedures in other databases. It extends SQL with loops, conditionals, and return types. Though its syntax may be strange to many developers it is much faster than anything running on the application server because the overhead of connecting to the database is eliminated, which is particularly useful when you would otherwise need to execute a query, wait for the result, and submit another query.

Though many other procedural languages exist for PostgreSQL, such as PL/Python, PL/Perl, and PLV8, PL/pgSQL is a common starting point for developers who want to write their first PostgreSQL function because its syntax builds on SQL. It is also similar to PL/SQL, Oracle's native procedural language, so any developer familiar with PL/SQL will find the language familiar, and any developer who intends to develop Oracle applications in the future but wants to start with a free database can transition from PL/pgSQL to PL/SQL with relative ease.

It should be emphasized that other procedural languages exist and PL/pgSQL is not necessarily superior to them in any way, including speed, but examples in PL/pgSQL can serve as a common reference point for other languages used for writing PostgreSQL functions. PL/pgSQL has the most tutorials and books of all the PLs and can be a springboard to learning the languages with less documentation.

Here are links to some free guides and books on PL/pgSQL:

- The official documentation:  
<https://www.postgresql.org/docs/current/static/plpgsql.html>
- w3resource.com tutorial: <http://www.w3resource.com/PostgreSQL/pl-pgsql-tutorial.php>
- postgres.cz tutorial: [http://postgres.cz/wiki/PL/pgSQL\\_\(en\)](http://postgres.cz/wiki/PL/pgSQL_(en))
- PostgreSQL Server Programming, 2nd Edition:  
<https://www.packtpub.com/big-data-and-business-intelligence/postgresql-server-programming-second-edition>
- PostgreSQL Developer's Guide: <https://www.packtpub.com/big-data-and-business-intelligence/postgresql-developers-guide>

## **PROGRAMMING WITH PL/pgSQL**

- Basic PL/pgSQL Function
- custom exceptions
- PL/pgSQL Syntax
- RETURNS Block

### **Overview**

1. Advantages of Using PL/pgSQL

2. Supported Argument and Result Data Types

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions, procedures, and triggers,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, procedures, and operators,
- can be defined to be trusted by the server,
- is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.



In PostgreSQL 9.0 and later, PL/pgSQL is installed by default. However it is still a loadable module, so especially security-conscious administrators could choose to remove it.

### 1.1. Advantages of Using PL/pgSQL

SQL is the language PostgreSQL and most other relational databases use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions.

Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.

### 1.2. Supported Argument and Result Data Types

Functions written in PL/pgSQL can accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as accepting record, which means that any composite type will do as input, or as returning record, which means that the result is a row type whose columns are determined by specification in the calling query, as discussed in Section 7.2.1.4.

PL/pgSQL functions can be declared to accept a variable number of arguments by using the VARIADIC marker. This works exactly the same way as for SQL functions, as discussed in Section 38.5.6.

PL/pgSQL functions can also be declared to accept and return the polymorphic types described in Section 38.2.5, thus allowing the actual data types handled by the function to vary from call to call. Examples appear in Section 43.3.1.

PL/pgSQL functions can also be declared to return a “set” (or table) of any data type that can be returned as a single instance. Such a function generates its output by executing RETURN NEXT for each desired element of the result set, or by using RETURN QUERY to output the result of evaluating a query.

Finally, a PL/pgSQL function can be declared to return void if it has no useful return value. (Alternatively, it could be written as a procedure in that case.) PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values. The RETURNS TABLE notation can also be used in place of RETURNS SETOF.

## 2. Structure of PL/pgSQL

Functions written in PL/pgSQL are defined to the server by executing CREATE FUNCTION commands. Such a command would normally look like, say, CREATE FUNCTION somefunc(integer, text) RETURNS integer AS 'function body text' LANGUAGE plpgsql;

The function body is simply a string literal so far as CREATE FUNCTION is concerned. It is often helpful to use dollar quoting (see Section 4.1.2.4) to write the function body, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function body must be escaped by doubling them. Almost all the examples in this chapter use dollar-quoted literals for their function bodies.

PL/pgSQL is a block-structured language. The complete text of a function body must be a *block*. A block is defined as:

```
[ <<label>> ]
[ DECLARE
  declarations ]
BEGIN
  statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

### Tip

A common mistake is to write a semicolon immediately after BEGIN. This is incorrect and will result in a syntax error.

A *label* is only needed if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. If a label is given after END, it must match the label at the block's beginning.

All key words are case-insensitive. Identifiers are implicitly converted to lower case unless double-quoted, just as they are in ordinary SQL commands.

Comments work the same way in PL/pgSQL code as in ordinary SQL. A double dash (--) starts a comment that extends to the end of the line. A /\* starts a block comment that extends to the matching occurrence of \*/. Block comments nest.

Any statement in the statement section of a block can be a *subblock*. Subblocks can be used for logical grouping or to localize variables to a small group of statements. Variables declared in a subblock mask any similarly-named variables of outer blocks for the duration of the subblock; but you can access the outer variables anyway if you qualify their names with their block's label. For example:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints
50
    END;

    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

## UNIT - III

### RELATIONAL DATABASE DESIGN AND NORMALIZATION

#### Syllabus

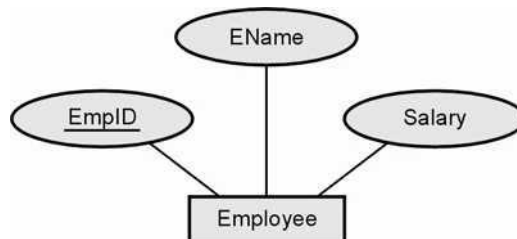
**ER-to-Relational Mapping – Update anomalies-Functional Dependencies – Inference rules-minimal cover-properties of relational decomposition- Normalization (upto BCNF).**

#### **ER to Relational Mapping**

In this section we will discuss how to map various ER model constructs to Relational Model construct.

##### **3.1.1 Mapping of Entity Set to Relationship**

- An entity set is mapped to a relation in a straightforward way.
- Each attribute of entity set becomes an attribute of the table.
- The primary key attribute of entity set becomes an entity of the table.
- For example - Consider following ER diagram.



The converted employee table is as follows -

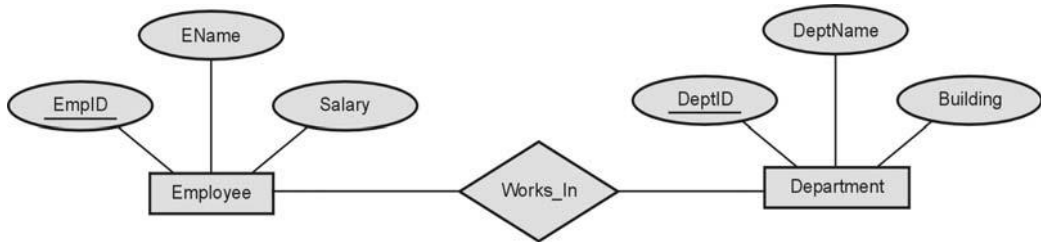
EmpID	EName	Salary
201	Poonam	30000
202	Ashwini	35000
203	Sharda	40000

The SQL statement captures the information for above ER diagram as follows -

```
CREATE TABLE Employee( EmpID CHAR(11),
  EName CHAR(30),
  Salary INTEGER,
  PRIMARY KEY(EmpID))
```

### 3.1.2 Mapping Relationship Sets(without Constraints) to Tables

- Create a table for the relationship set.
  - Add all primary keys of the participating entity sets as fields of the table.
  - Add a field for each attribute of the relationship.
  - Declare a primary key using all key fields from the entity sets.
  - Declare foreign key constraints for all these fields from the entity sets.
- For example - Consider following ER model



The SQL statement captures the information for relationship present in above ER diagram as follows -

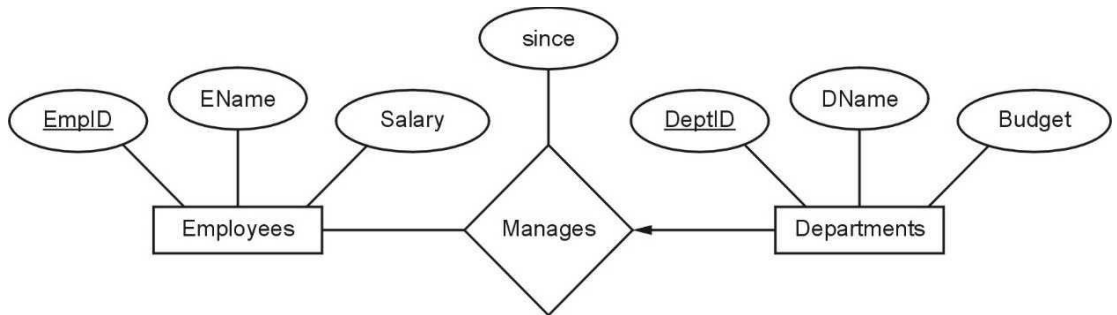
```
CREATE TABLE Works_In (EmpID CHAR(11),
                        DeptID CHAR(11),
                        EName CHAR(30),
                        Salary INTEGER,
                        DeptName CHAR(20),
                        Building CHAR(10),
                        PRIMARY KEY(EmpID,DeptID),
                        FOREIGN KEY (EmpID) REFERENCES Employee,
                        FOREIGN KEY (DeptID) REFERENCES Department
                        )
```

### 3.1.3 Mapping Relationship Sets( With Constraints) to Tables

- If a relationship set involves **n** entity sets and some **m** of them are linked **via arrows** in the ER diagram, the key for anyone of these **m** entity sets **constitutes a key** for the relation to which the relationship set is mapped.
- Hence we have **m** candidate keys, and one of these should be designated as the **primary key**.
- There are two approaches used to convert a relationship sets with key constraints into table.

- **Approach 1 :**

- By this approach the relationship associated with more than one entities is separately represented using a table. For example - Consider following ER diagram. Each **Dept has at most one manager, according to the key constraint on Manages.**



Here the constraint is each department has at the most one manager to manage it. Hence no two tuples can have same DeptID. Hence there can be a separate table named **Manages** with DeptID as Primary Key. The table can be defined using following SQL statement

```
CREATE TABLE Manages(EmpID CHAR(11),
                    DeptID INTEGER,
                    Since DATE,
                    PRIMARY KEY(DeptID),
                    FOREIGN KEY (EmpID) REFERENCES Employees,
                    FOREIGN KEY (DeptID) REFERENCES Departments)
```

- **Approach 2 :**

- In this approach , it is preferred **to translate a relationship set with key constraints.**
- It is a superior approach because, it avoids creating a distinct table for the relationship set.
- The idea is to include the information about the relationship set in the table corresponding to the entity set with the key, taking advantage of the key constraint.
- This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations.

- The only drawback to this approach is that space could be wasted if several departments have no managers.
- The following SQL statement, defining a **Dep\_Mgr** relation that captures the information in both **Departments** and **Manages**, illustrates the second approach to translating relationship sets with key constraints :

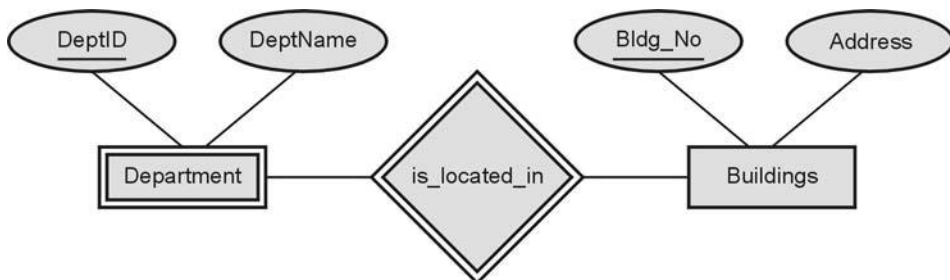
```
CREATE TABLE Dep_Mgr ( DeptID INTEGER,
                      DName CHAR(20),
                      Budget REAL,
                      EmpID CHAR (11),
                      since DATE,
                      PRIMARY KEY (DeptID),
                      FOREIGN KEY (EmpID) REFERENCES Employees)
```

### 3.1.4 Mapping Weak Entity Sets to Relational Mapping

A weak entity can be identified uniquely only by considering the primary key of another (owner) entity. Following steps are used for mapping Weka Entity Set to Relational Mapping

- Create a table for the weak entity set.
- Make each attribute of the weak entity set a field of the table.
- Add fields for the primary key attributes of the identifying owner.
- Declare a foreign key constraint on these identifying owner fields.
- Instruct the system to automatically delete any tuples in the table for which there are no owners

For example - Consider following ER model

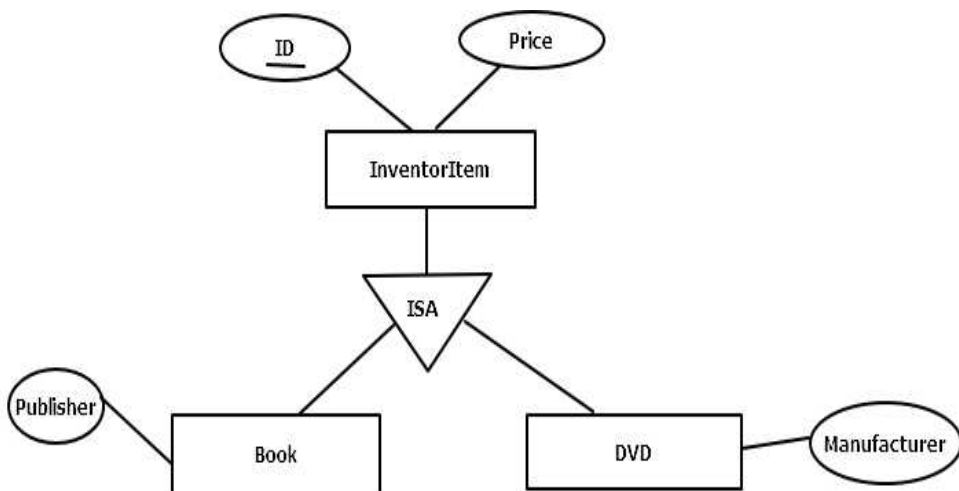


Following SQL Statement illustrates this mapping

```
CREATE TABLE Department(DeptID CHAR(11),
    DeptName CHAR(20),
    Bldg_No CHAR(5),
    PRIMARY KEY (DeptID,Bldg_No),
    FOREIGN KEY(Bldg_No) References Buildings on delete cascade
)
```

### Mapping of Specialization / Generalization(EER Construct) to Relational Mapping

The specialization/Generalization relationship(Enhanced ER Construct) can be mapped to database tables(relations) using three methods. To demonstrate the methods, we will take the – InventoryItem, Book, DVD



**Method 1 :** All the entities in the relationship are mapped to individual tables

```
InventoryItem(ID, name)
Book(ID,Publisher)
DVD(ID, Manufacturer)
```

**Method 2 :** Only subclasses are mapped to tables. The attributes in the superclass are duplicated in all subclasses. For example -

```
Book(ID,name,Publisher)
DVD(ID, name,Manufacturer)
```



**Method 3** : Only the superclass is mapped to a table. The attributes in the subclasses are taken to the superclass. For example -

InventoryItem(ID , name,Publisher,Manufacturer)

This method will introduce **null** values. When we insert a **Book** record in the table, the **Manufacturer** column value will be null. In the same way, when we insert a **DVD** record in the table, the **Publisher** value will be null.

### 3.2 Concept of Relational Database Design

- There are two primary goals of relational database design – i) to generate a set of relation schemas that allows us to store information without unnecessary redundancy, and ii) to allows us to retrieve information easily.
- For achieving these goals, the database design need to be **normalized**. That means we have to check whether the schema is it normal form or not.
- For checking the normal form of the schema, it is necessary to check the functional dependencies and other data dependencies that exists within the schema.

Hence before letting us know what the normalization means, it is necessary to understand the concept of functional dependencies.

### 3.3 Functional Dependencies

**Definition** : Let P and Q be sets of columns, then: P functionally determines Q, written  $P \rightarrow Q$  if and only if any two rows that are equal on (all the attributes in) P must be equal on (all the attributes in) Q.

In other words, the functional dependency holds if

$$T1.P = T2.P, \text{ then } T1.Q=T2.Q$$

Where notation T1.P projects the tuple T1 onto the attribute in P.

**For example** : Consider a relation in which the roll of the student and his/her name is stored as follows :

R	N
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

**Fig. 3.8.1** : Table which holds functional dependency i.e. R->B

Here,  $R \rightarrow N$  is true. That means the functional dependency holds true here. Because for every assigned RollNumber of student there will be unique name. For instance : The name of the Student whose RollNo is 1 is AAA. But if we get two different names for the same roll number then that means the table does not hold the functional dependency. Following is such table –

R	N
1	AAA
2	BBB
3	CCC
1	XXX
2	YYY

**Fig. 3.8.2 : Table which does not hold functional dependency**

In above table for RollNumber 1 we are getting two different names - “AAA” and “XXX”. Hence here it does not hold the functional dependency.

### **3.8.1 Computing Closure Set of Functional Dependency (Armstrong’s Axioms)**

The closure set is a set of all functional dependencies implied by a given set F. It is denoted by  $F^+$

The closure set of functional dependency can be computed using basic three rules which are also called as Armstrong’s Axioms.

These are as follows -

- i) Reflexivity :** If  $X \supseteq Y$ , then  $X \rightarrow Y$
- ii) Augmentation :** If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$  for any Z
- iv) Transitivity :** If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$

In addition to above axioms some additional rules for computing closure set of functional dependency are as follows -

- **Union :** If  $X \rightarrow Y$  and  $X \rightarrow Z$  then  $X \rightarrow YZ$
- **Decomposition :** If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$

**Example 3.8.1** Compute the closure of the following set of functional dependencies for a relation scheme  $R(A,B,C,D,E)$ ,  $F=\{A \rightarrow BC, CD \rightarrow E, B \rightarrow D, E \rightarrow A\}$

**Solution :** Consider F as follows

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

The closure can be written for each attribute of relation as follows

- $(A)^+ =$  **Step 1 :**  $\{A\}$   $\rightarrow$  the attribute itself  
**Step 2 :**  $\{ABC\}$  as  $A \rightarrow BC$   
**Step 3 :**  $\{ABCD\}$  as  $B \rightarrow D$   
**Step 4 :**  $\{ABCDE\}$  as  $CD \rightarrow E$   
**Step 5 :**  $\{ABCDE\}$  as  $E \rightarrow A$  and  $A$  is already present  
Hence  $(A)^+ = \{ABCDE\}$
- $(B)^+ =$  **Step 1:**  $\{B\}$   
**Step 2 :**  $\{BD\}$  as  $B \rightarrow D$   
**Step 3 :**  $\{BD\}$  as there is no  $BD$  pair on LHS of F  
Hence  $(B)^+ = \{BD\}$
- $(C)^+ =$  **Step 1:**  $\{C\}$   
**Step 2 :**  $\{C\}$  as there is no single  $C$  on LHS of F  
Hence  $(C)^+ = \{C\}$
- $(D)^+ =$  **Step 1 :**  $\{D\}$   
**Step 3 :**  $\{D\}$  as there is no  $BD$  pair on LHS of F  
Hence  $(D)^+ = \{D\}$
- $(E)^+ =$  **Step 1 :**  $\{E\}$   
**Step 2 :**  $\{EA\}$  as  $E \rightarrow A$   
**Step 3 :**  $\{EABC\}$  as  $A \rightarrow BC$   
**Step 4 :**  $\{EABCD\}$  as  $B \rightarrow D$   
**Step 5 :**  $\{EABCD\}$  as  $CD \rightarrow E$  and  $E$  is already present

By rearranging we get {ABCDE}

Hence  $(E)^+ = \{ABCDE\}$

- $(CD)^+ =$  **Step 1:**{CD}

**Step 2 :**{CDE}

**Step 3 :**{CDEA}

**Step 4 :**{CDEAB}

By rearranging we get {ABCDE}

Hence  $(CD)^+ = \{ABCDE\}$

**Example 3.8.2** Compute the closure of the following set of functional dependencies for a relation scheme  $R(A,B,C,D,E)$ ,  $F=\{A \rightarrow BC, CD \rightarrow E, B \rightarrow D, E \rightarrow A\}$  and Find the candidate key.

**Solution :** For finding the closure of functional dependencies - Refer example 2.8.1.

We can identify candidate from the given relation schema with the help of functional dependency. For that purpose we need to compute the closure set of attribute. Now we will find out the closure set which can completely identify the relation  $R(A,B,C,D)$ .

Let,

- $(A)^+ = \{ABCDE\}$
- $(B)^+ = \{BD\}$
- $(C)^+ = \{C\}$
- $(D)^+ = \{D\}$
- $(E)^+ = \{ABCDE\}$
- $(CD)^+ = \{ABCDE\}$

Clearly, only  $(A)^+, (E)^+$  and  $(CD)^+$  gives us {ABCD} i.e. complete relation R. Hence these are the candidate keys.

### **3.8.2 Canonical Cover or Minimal Cover**

**Formal Definition :** A minimal cover for a set F of FDs is a set G of FDs such that :

- 1) Every dependency in G is of the form  $X \rightarrow A$ , where A is a single attribute.
- 2) The **closure  $F^+$**  is equal to the **closure  $G^+$** .
- 3) If we obtain a set H of dependencies from G by deleting one or more dependencies or by deleting attributes from a dependency in G, then  $F^+ \neq H^+$ .

## Concept of Extraneous Attributes

**Definition :** An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies. The formal definition of extraneous attributes is as follows:

Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$

- Attribute  $A$  is extraneous in  $\alpha$  if  $A \in \alpha$ , and  $F$  logically implies  $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$
- Attribute  $A$  is extraneous in  $\beta$  if  $A \in \beta$  and the set of functional dependencies  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$  logically implies  $F$ .

## Algorithm for computing Canonical Cover for set of functional Dependencies $F$

$F_c = F$

**repeat**

Use the union rule to replace any dependencies in  $F_c$  of the form

$\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  and  $\alpha_1 \rightarrow \beta_1\beta_2$

Find a functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  with an extraneous attribute either in  $\alpha$  or in  $\beta$ .

/\* The test for extraneous attributes is done using  $F_c$ , not  $F$  \*/

If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$  in  $F_c$ .

**until** ( $F_c$  does not change)

**Example 3.8.3** Consider the following functional dependencies over the attribute set  $R(ABCDE)$  for finding minimal cover  $FD = \{A \rightarrow C, AC \rightarrow D, B \rightarrow ADE\}$

**Solution :**

**Step 1 : Split the FD** such that R.H.S contain single attribute. Hence we get

$A \rightarrow C$

$AC \rightarrow D$

$B \rightarrow A$

$B \rightarrow D$

$B \rightarrow E$

**Step 2 :** Find the **redundant entries** and **delete** them. This can be done as follows -

- **For  $A \rightarrow C$  :** We find  $(A)^+$  by assuming that we delete  $A \rightarrow C$  temporarily. We get  $(A)^+ = \{A\}$ . Thus from  $A$  it is not possible to obtain  $C$  by deleting  $A \rightarrow C$ . This means we can not delete  $A \rightarrow C$

- **For AC->D :** We find  $(AC)^+$  by assuming that we delete AC->D temporarily. We get  $(AC)^+ = \{AC\}$ . Thus by such deletion it is not possible to obtain D. This means we can not delete AC->D
- **For B->A :** We find  $(B)^+$  by assuming that we delete B->A temporarily. We get  $(B)^+ = \{BDE\}$ . Thus by such deletion it is not possible to obtain A. This means we can not delete B->A
- **For B->D :** We find  $(B)^+$  by assuming that we delete B->D temporarily. We get  $(B)^+ = \{BEACD\}$ . This shows clearly that even if we delete B->D we can obtain D. This means we can delete B->A. Thus it is redundant.
- **For B->E :** We find  $(B)^+$  by assuming that we delete B->E temporarily. We get  $(B)^+ = \{BDAC\}$ . Thus by such deletion it is not possible to obtain E. This means we can not delete B->E

To summarize we get now

A->C

AC->D

B->A

B->E

Thus R.H.S gets simplified.

**Step 3 :** Now we will simplify L.H.S.

Consider AC->D. Here we can split A and C. For that we find closure set of A and C.

$$(A)^+ = (AC)$$

$$(C)^+ = (C)$$

Thus C can be obtained from both A as well as C. That also means we need not have to have AC on L.H.S. Instead, only A can be allowed and C can be eliminated. Thus after simplification we get

A->D

To summarize we get now

A->C

A->D

B->A

B->E

Thus L.H.S gets simplified.

**Step 3 :** The simplified L.H.S. and R.H.S can be combined together to form

A->CD

B->AE

This is a **minimal cover** or **Canonical cover** of functional dependencies.

### 3.4 Concept of Redundancy and Anomalies

**Definition :** Redundancy is a condition created in database in which same piece of data is held at two different places.

Redundancy is at the root of several problems associated with relational schemas.

**Problems caused by redundancy :** Following problems can be caused by redundancy-

- i) **Redundant storage :** Some information is stored repeatedly.
- ii) **Update anomalies :** If one copy of such repeated data is updated then inconsistency is created unless all other copies are similarly updated.
- iii) **Insertion anomalies :** Due to insertion of new record repeated information get added to the relation schema.
- iv) **Deletion anomalies :** Due to deletion of particular record some other important information associated with the deleted record get deleted and thus we may lose some other important information from the schema.

**Example :** Following example illustrates the above discussed anomalies or redundancy problems

Consider following Schema in which all possible information about Employee is stored.

EmpID	EName	Salary	DeptID	DeptName	DeptLoc
1	AAA	10000	101	XYZ	Pune
2	BBB	20000	101	XYZ	Pune
3	CCC	30000	101	XYZ	Pune
4	DDD	40000	102	PQR	Mumbai

Redundancy!!!

- 1) **Redundant storage :** Note that the information about **DeptID**, **DeptName** and **DeptLoc** is repeated.
- 2) **Update anomalies :** In above table if we change **DeptLoc** of Pune to Chennai, then it will result **inconsistency** as for DeptID 101 the DeptLoc is Pune. Or otherwise, we need to **update multiple copies** of **DeptLoc** from Pune to Chennai. Hence this is an update anomaly.
- 3) **Insertion anomalies :** For above table if we want to add new tuple say (5, EEE,50000) for **DeptID 101** then it will cause repeated information of

(101, XYZ,Pune) will occur.

- 4) **Deletion anomalies** : For above table, if we delete a record for **EmpID 4**, then automatically information about the **DeptID 102,DeptName PQR** and **DeptLoc Mumbai** will get deleted and one may not be aware about **DeptID 102**. This causes deletion anomaly.

### 3.10 Decomposition

AU : Dec.-17, Marks 7

- Decomposition is the process of breaking down one table into multiple tables.
- **Formal definition of decomposition is -**
- A decomposition of relation Schema R consists of replacing the relation Schema by two relation schema that each contain a subset of attributes of R and together include all attributes of R by storing projections of the instance.

For example - Consider the following table

**Employee\_Department** table as follows -

Eid	Ename	Age	City	Salary	Deptid	DeptName
E001	ABC	29	Pune	20000	D001	Finance
E002	PQR	30	Pune	30000	D002	Production
E003	LMN	25	Mumbai	5000	D003	Sales
E004	XYZ	24	Mumbai	4000	D004	Marketing
E005	STU	32	Hyderabad	25000	D005	Human Resource

We can decompose the above relation Schema into two relation schemas as **Employee (Eid, Ename, Age, City, Salary)** and **Department (Deptid, Eid, DeptName)**. as follows -

**Employee Table**

Eid	Ename	Age	City	Salary
E001	ABC	29	Pune	20000
E002	PQR	30	Pune	30000
E003	LMN	25	Mumbai	5000
E004	XYZ	24	Mumbai	4000
E005	STU	32	Hyderabad	25000



## Department Table

Deptid	Eid	DeptName
D001	E001	Finance
D002	E002	Production
D003	E003	Sales
D004	E004	Marketing
D005	E005	Human Resource

- The decomposition is used for eliminating redundancy.
- **For example :** Consider following relation **Schema R** in which we assume that the grade determines the salary, the redundancy is caused

## Schema R

Name	eid	deptname	Grade	Salary
AAA	121	Accounts	2	8000
AAA	132	Sales	3	7000
BBB	101	Marketing	4	7000
CCC	106	Purchase	2	8000

**Redundancy!!!**

- Hence, the above table can be decomposed into two Schema S and T as follows :

Schema S			
Name	eid	deptname	Grade
AAA	121	Accounts	2
AAA	132	Sales	3
BBB	101	Marketing	4
CCC	106	Purchase	2

Schema T	
Grade	Salary
2	8000
3	7000
4	7000
2	8000

## Problems Related to Decomposition :

Following are the potential problems to consider :

- 1) Some queries become more **expensive**.
- 2) Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
- 3) Checking some dependencies may require joining the instances of the decomposed relations.
- 4) There may be loss of information during decomposition.

## Properties Associated With Decomposition

There are two properties associated with decomposition and those are –

- 1) **Loss-less Join or non Loss Decomposition** : When all information found in the original database is preserved after decomposition, we call it as loss less or non loss decomposition.
- 2) **Dependency Preservation** : This is a property in which the constraints on the original table can be maintained by simply enforcing some constraints on each of the smaller relations.

### 3.10.1 Non-loss Decomposition or Loss-less Join

The lossless join can be defined using following three conditions :

- i) **Union** of attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.

$$\text{Att}(R1) \cup \text{Att}(R2) = \text{Att}(R)$$

- ii) **Intersection** of attributes of R1 and R2 must not be NULL.

$$\text{Att}(R1) \cap \text{Att}(R2) \neq \Phi$$

- iii) **Common attribute** must be a key for at least one relation (R1 or R2)

$$\text{Att}(R1) \cap \text{Att}(R2) \rightarrow \text{Att}(R1)$$

or  $\text{Att}(R1) \cap \text{Att}(R2) \rightarrow \text{Att}(R2)$

**Example 3.10.1** Consider the following relation  $R(A,B,C,D)$  and FDs  $A \rightarrow BC$ , is the decomposition of R into  $R1(A,B,C)$ ,  $R2(A,D)$ . Check if the decomposition is lossless join or not.

**Solution :**

**Step 1 :** Here  $\text{Att}(R1) \cup \text{Att}(R2) = \text{Att}(R)$  i.e  $R1(A,B,C) \cup R2(A,D) = (A,B,C,D)$  i.e R. Thus first condition gets satisfied.

**Step 2 :** Here  $R1 \cap R2 = \{A\}$ . Thus  $\text{Att}(R1) \cap \text{Att}(R2) \neq \emptyset$ . Here the second condition gets satisfied.

**Step 3 :**  $\text{Att}(R1) \cap \text{Att}(R2) \rightarrow \{A\}$ . Now  $(A)^+ = \{A,B,C\} \in$  attributes of R1. Thus the third condition gets satisfied.

This shows that the given decomposition is a **lossless join**.

**Example 3.10.2** Consider the following relation  $R(A,B,C,D,E,F)$  and FDs  $A \rightarrow BC$ ,  $C \rightarrow A$ ,  $D \rightarrow E$ ,  $F \rightarrow A$ ,  $E \rightarrow D$  is the decomposition of R into  $R1(A,C,D)$ ,  $R2(B,C,D)$ , and  $R3(E,F,D)$ . Check for lossless.

**Solution :**

**Step 1 :**  $R1 \cup R2 \cup R3 = R$ . Here the first condition for checking lossless join is satisfied as  $(A,C,D) \cup (B,C,D) \cup (E,F,D) = \{A,B,C,D,E,F\}$  which is nothing but R.

**Step 2 :** Consider  $R1 \cap R2 = \{CD\}$  and  $R2 \cap R3 = \{D\}$ . Hence second condition of intersection not being  $\emptyset$  gets satisfied.

**Step 3 :** Now, consider  $R1(A,C,D)$  and  $R2(B,C,D)$ . We find  $R1 \cap R2 = \{CD\}$

$(CD)^+ = \{ABCDE\} \in$  attributes of  $R1$  i.e.  $\{A,C,D\}$ . Hence condition 3 for checking lossless join for  $R1$  and  $R2$  gets satisfied.

**Step 4 :** Now, consider  $R2(B,C,D)$  and  $R3(E,F,D)$ . We find  $R2 \cap R3 = \{D\}$ .

$(D)^+ = \{D,E\}$  which is neither complete set of attributes of  $R2$  or  $R3$ . [Note that  $F$  is missing for being attribute of  $R3$ ].

Hence it is not **lossless join decomposition**. Or in other words we can say it is a **lossy decomposition**.

**Example 3.10.3** Suppose that we decompose schema  $R=(A,B,C,D,E)$  into  $(A,B,C)$   $(C,D,E)$   
Show that it is not a lossless decomposition.

**Solution :Step 1 :** Here we need to assume some data for the attributes  $A, B, C, D,$  and  $E$ . Using this data we can represent the relation as follows –

**Relation R**

A	B	C	D	E
a	1	x	p	q
b	2	x	r	s

**Relation R1 = (A,B,C)**

A	B	C
a	1	x
b	2	x

**Relation R2 = (C,D,E)**

C	D	E
x	p	q
X	r	s

**Step 2 :** Now we will join these tables using natural join, i.e. the join based on common attribute  $C$ . We get  $R1 \bowtie R2$  as

A	B	C	D	E
a	1	x	p	q
a	1	x	r	s
b	2	x	p	q
b	2	x	r	s

Here we get more rows or tuples than original relation R

Clearly  $R1 \bowtie R2 \notin R$ . Hence it is not lossless decomposition.

### 3.10.2 Dependency Preservation

- **Definition** : A Decomposition  $D = \{R1, R2, R3, \dots, Rn\}$  of  $R$  is dependency preserving for a set  $F$  of Functional dependency if  $(F1 \cup F2 \cup \dots \cup Fm)^+ = F^+$ .
- If decomposition is not dependency-preserving, some dependency is lost in the decomposition.

**Example 3.10.4** Consider the relation  $R(A, B, C)$  for functional dependency set  $\{A \rightarrow B$  and  $B \rightarrow C\}$  which is decomposed into two relations  $R1 = (A, C)$  and  $R2 = (B, C)$ . Then check if this decomposition dependency preserving or not.

**Solution** : This can be solved in following steps :

**Step 1** : For checking whether the decomposition is dependency preserving or not we need to check

following condition

$$F^+ = (F1 \cup F2)^+$$

**Step 2** : We have with us the  $F^+ = \{ A \rightarrow B$  and  $B \rightarrow C \}$

**Step 3** : Let us find  $(F1)^+$  for relation  $R1$  and  $(F2)^+$  for relation  $R2$

R1(A,C)	
A→A	Trivial
C→C	Trivial
A→C	= In (F) <sup>+</sup> A→B→C and it is Nontrivial
AC→AC	Trivial
A→B	but is not useful as B is not part of R1 set
We can not obtain C→A	

R2(B,C)	
B→B	Trivial
C→C	Trivial
B→C	= In (F) <sup>+</sup> B→C and it is Non-Trivial
BC→BC	Trivial
We can not obtain C→B	

**Step 4** : We will eliminate all the trivial relations and useless relations. Hence we can obtain  $R1$  and  $R2$  as

R1(A,C)	
A→C	Nontrivial

R2(B,C)	
B→C	Non-Trivial

$$(F1 \cup F2)^+ = \{A \rightarrow C, B \rightarrow C\} \neq \{A \rightarrow B, B \rightarrow C\} \text{ i.e. } (F)^+$$

Thus the condition specified in step 1 i.e.  $F^+ = (F1 \cup F2)^+$  is **not true**. Hence it is **not dependency preserving decomposition**.

**Example 3.10.5** Let relation  $R(A,B,C,D)$  be a relational schema with following functional dependencies  $\{A \rightarrow B, B \rightarrow C, C \rightarrow D, \text{ and } D \rightarrow B\}$ . The decomposition of  $R$  into  $(A,B)$ ,  $(B,C)$  and  $(B,D)$ . Check whether this decomposition is dependency preserving or not.

**Solution :**

**Step 1 :** Let  $(F)^+ = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow B\}$ .

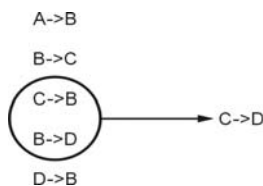
**Step 2 :** We will find  $(F1)^+$ ,  $(F2)^+$ ,  $(F3)^+$  for relations  $R1(A,B)$ ,  $R2(B,C)$  and  $R3(B,D)$  as follows -

R1(A,B)	R2(B,C)	R3(B,D)
A → A Trivial	B → B Trivial	B → B Trivial
B → B Trivial	C → C Trivial	D → D Trivial
A → B ∴ (F) <sup>+</sup> and it's non Trivial	B → C ∴ (F) <sup>+</sup> and it's non Trivial	B → D ∴ (F) <sup>+</sup> as and B → C → D and it's non Trivial
B → A can not be obtained	C → B ∴ In (F) <sup>+</sup> and C → D → C and it is Nontrivial	D → B ∴ (F) <sup>+</sup> and it's non Trivial
AB → AB	BC → BC Trivial	BD → BD Trivial

**Step 3 :** We will eliminate all the trivial relations and useless relations. Hence we can obtain  $R1 \cup R2 \cup R3$  as

R1(A,B) A → B	R2(B,C) B → C C → B	R3(B,D) B → D D → B
------------------	---------------------------	---------------------------

**Step 4 :** As from above FD's we get



**Step 5 :** This proves that  $F^+ = (F1 \cup F2 \cup F3)^+$ . Hence given **decomposition is dependency preserving**.

## 3.11 Normal Forms

AU : Dec.-14, 15, May-18, Marks 16

- Normalization is the process of reorganizing data in a database so that it meets **two basic requirements**:
  - 1) There is **no redundancy** of data (all data is stored in only one place), and
  - 2) **data dependencies** are logical (all related data items are stored together)
- The normalization is important because it allows database to take up **less disk space**.
- It also help in increasing the **performance**.

### 3.11.1 First Normal Form

The table is said to be in 1NF if it follows following rules -

- i) It should only have single (atomic) valued attributes/columns.
- ii) Values stored in a column should be of the same domain
- iii) All the columns in a table should have unique names.
- iv) And the order in which data is stored, does not matter.

Consider following Student table

**Student**

sid	sname	Phone
1	AAA	11111 22222
2	BBB	33333
3	CCC	44444 55555

As there are multiple values of phone number for sid 1 and 3, the above table is not in 1NF. We can make it in 1NF. The conversion is as follows -

sid	sname	Phone
1	AAA	11111
1	AAA	22222
2	BBB	33333
3	CCC	44444
3	CCC	55555

### 3.11.2 Second Normal Form

Before understanding the second normal form let us first discuss the concept of partial functional dependency and prime and non prime attributes.

#### Concept of Partial Functional Dependency

Partial dependency means that a nonprime attribute is functionally dependent on part of a candidate key.

For example : Consider a relation  $R(A,B,C,D)$  with functional dependency  $\{AB \rightarrow CD, A \rightarrow C\}$

Here  $(AB)$  is a candidate key because

$$(AB)^+ = \{ABCD\} = \{R\}$$

Hence  $\{A,B\}$  are prime attributes and  $\{C,D\}$  are non prime attribute. In  $A \rightarrow C$ , the non prime attribute  $C$  is dependent upon  $A$  which is actually a part of candidate key  $AB$ . Hence due to  $A \rightarrow C$  we get partial functional dependency.

#### Prime and Non Prime Attributes

- **Prime attribute** : An attribute, which is a part of the candidate-key, is known as a prime attribute.
- **Non-prime attribute** : An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.
- **Example** : Consider a Relation  $R=\{A,B,C,D\}$  and candidate key as  $AB$ , the Prime attributes :  $A, B$

Non Prime attributes :  $C, D$

#### The Second Normal Form

For a table to be in the Second Normal Form, following conditions must be followed

- i) It should be in the First Normal form.
- ii) It should not have partial functional dependency.

For example : Consider following table in which every information about a the Student is maintained in a table such as student id(sid), student name(sname), course id(cid) and course name(cname).

#### Student\_Course

sid	sname	cid	cname
1	AAA	101	C
2	BBB	102	C++
3	CCC	101	C
4	DDD	103	Java

This table is not in 2NF. For converting above table to 2NF we must follow the following steps -

**Step 1 :** The above table is in 1NF.

**Step 2 :** Here **sname** and **sid** are associated similarly **cid** and **cname** are associated with each other. Now if we delete a record with **sid=2**, then automatically the course C++ will also get deleted. Thus,

**sid->sname** or **cid->cname** is a partial functional dependency, because **{sid,cid}** should be essentially a candidate key for above table. Hence to bring the above table to 2NF we must decompose it as follows :

### Student

sid	sname	cid
1	AAA	101
2	BBB	102
3	CCC	101
4	DDD	103

Here candidate key is (sid,cid) and (sid,cid)->sname

### Course

cid	cname
101	C
102	C++
101	C
103	Java

Here candidate key is cid Here cid->cname

Thus now table is in 2NF as there is no partial functional dependency

### 3.11.3 Third Normal Form

Before understanding the third normal form let us first discuss the concept of transitive dependency, super key and candidate key

#### Concept of Transitive Dependency

A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies. For example -

X -> Z is a transitive dependency if the following functional dependencies hold true :

X->Y

Y->Z



## Concept of Super key and Candidate Key

**Superkey** : A super key is a set or one of more columns (attributes) to uniquely identify rows in a table.

**Candidate key** : The minimal set of attribute which can uniquely identify a tuple is known as candidate key. For example consider following table

RegID	RollNo	Sname
101	1	AAA
102	2	BBB
103	3	CCC
104	4	DDD

### Superkeys

- {RegID}
- {RegID, RollNo}
- {RegID,Sname}
- {RollNo,Sname}
- {RegID, RollNo,Sname}

### Candidate Keys

- {RegID}
- {RollNo}

## Third Normal Form

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

- i) It is in the Second Normal form.(i.e. it does not have partial functional dependency)
- ii) It doesn't have transitive dependency.

Or in other words

In other words 3NF can be defined as : A table is in 3NF if it is in 2NF and for each functional dependency

$X \rightarrow Y$

at least one of the following conditions hold :

- i) X is a super key of table
- ii) Y is a prime attribute of table

For example : Consider following table **Student\_details** as follows -

sid	sname	zipcode	cityname	state
1	AAA	11111	Pune	Maharashtra
2	BBB	22222	Surat	Gujarat
3	CCC	33333	Chennai	Tamilnadu
4	DDD	44444	Jaipur	Rajasthan
5	EEE	55555	Mumbai	Maharashtra

Here

**Super keys** : {sid},{sid,sname},{sid,sname,zipcode}, {sid,zipcode,cityname}... and so on.

**Candidate keys** : {sid}

**Non-Prime attributes** : {sname,zipcode,cityname,state}

The dependencies can be denoted as

sid->sname

sid->zipcode

zipcode->cityname

cityname->state

The above denotes the transitive dependency. Hence above table is not in 3NF. We can convert it into 3NF as follows :

**Student**

sid	sname	zipcode
1	AAA	11111
2	BBB	22222
3	CCC	33333
4	DDD	44444
5	EEE	55555

**Zip**

zipcode	cityname	state
11111	Pune	Maharashtra
22222	Surat	Gujarat
33333	Chennai	Tamilnadu
44444	Jaipur	Rajasthan
55555	Mumbai	Maharashtra

**Example 3.11.1** Consider the relation  $R = \{A, B, C, D, E, F, G, H, I, J\}$  and the set of functional dependencies  $F = \{A, B \rightarrow C, A \rightarrow \{D, E\}, B \rightarrow F, F \rightarrow \{G, H\}, D \rightarrow \{I, J\}\}$

1. What is the key for  $R$ ? Demonstrate it using the inference rules.
2. Decompose  $R$  into 2NF, then 3NF relations.

**Solution :** Let,

$$A \rightarrow DE \text{ (given)}$$

$$\therefore A \rightarrow D, A \rightarrow E$$

$$\text{As } D \rightarrow IJ, A \rightarrow IJ$$

Using union rule we get

$$A \rightarrow DEIJ$$

$$\text{As } A \rightarrow A$$

$$\text{we get } A \rightarrow ADEIJ$$

Using augmentation rule we compute  $AB$

$$AB \rightarrow ABDEIJ$$

$$\text{But } AB \rightarrow C \text{ (given)}$$

$$\therefore AB \rightarrow ABCDEIJ$$

$$B \rightarrow F \text{ (given)} \quad F \rightarrow GH \quad \therefore B \rightarrow GH \text{ (transitivity)}$$

$$\therefore AB \rightarrow AGH \text{ is also true}$$

$$\text{Similarly } AB \rightarrow AF \quad \therefore B \rightarrow F \text{ (given)}$$

Thus now using union rule

$$AB \rightarrow ABCDEFGHIJ$$

$\therefore AB$  is a key

The table can be converted to 2NF as

$$R_1 = (\underline{A}, \underline{B}, C)$$

$$R_2 = (\underline{A}, D, E, I, J)$$

$$R_3 = (\underline{B}, F, G, H)$$

The above 2NF relations can be converted to 3NF as follows

$$R_1 = (\underline{A}, \underline{B}, C)$$

$$R_2 = (\underline{A}, D, E)$$

$$R_3 = (\underline{D}, I, J)$$

$$R_4 = (\underline{B}, E)$$

$$R_5 = (E, G, H).$$

### University Questions

1. What is database normalization ? Explain the first normal form, second normal form and third normal form.

**AU : May-18, Marks 13; Dec-15, Marks 16**

2. What are normal forms. Explain the types of normal form with an example.

**AU : Dec.-14, Marks 16**

### 3.12 Boyce / Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a **higher version** of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF.

A 3NF table which **does not have multiple overlapping** candidate keys is said to be in BCNF.

Or in other words,

For a table to be in BCNF, following conditions must be satisfied :

- i) R must be in 3rd Normal Form
- ii) For each functional dependency (  $X \rightarrow Y$  ), X should be a super Key. In simple words if Y is a prime attribute then X can not be non prime attribute.

For example - Consider following table that represents that a Student enrollment for the course -

#### Enrollment Table

sid	course	Teacher
1	C	Ankita
1	Java	Poonam
2	C	Ankita
3	C++	Supriya
4	C	Archana

From above table following observations can be made :

- One student can enroll for multiple courses. For example student with sid=1 can enroll for C as well as Java.
- For each course, a teacher is assigned to the student.
- There can be multiple teachers teaching one course for example course C can be taught by both the teachers namely - Ankita and Archana.
- The candidate key for above table can be (sid,course), because using these two columns we can find
- The above table holds following dependencies
  - (sid,course)->Teacher
  - Teacher->course
- The above table is not in BCNF because of the dependency **teacher->course**. Note that the teacher is not a superkey or in other words, **teacher** is a non prime attribute and **course** is a prime attribute and non-prime attribute derives the prime attribute.
- To convert the above table to BCNF we must decompose above table into Student and Course tables

### Student

sid	Teacher
1	Ankita
1	Poonam
2	Ankita
3	Supriya
4	Archana

### Course

Teacher	course
Ankita	C
Poonam	Java
Ankita	C
Supriya	C++
Archana	C

Now the table is in BCNF

**Example 3.12.1** Consider a relation(A,B,C,D) having following FDs.{AB->C, AB->D, C->A, B->D}. Find out the normal form of R.

**Solution :**

**Step 1 :** We will first find out the candidate key from the given FD.

$$(AB)^+ = \{ABCD\} = R$$

$$(BC)^+ = \{ABCD\} = R$$

$$(AC)^+ = \{AC\} \neq R$$

There is no involvement of D on LHS of the FD rules. Hence D can not be part of any candidate key. Thus we obtain two candidate keys **(AB)<sup>+</sup>** and **(BC)<sup>+</sup>**. Hence

**prime attributes** = {A,B,C}

**Non prime attributes** = {D}

**Step 2 :** Now, we will start checking from reverse manner, that means from BCNF, then 3NF, then 2NF.

**Step 3 :** For R being in BCNF for X->Y the X should be candidate key or super key.

From above FDs consider C->D in which C is not a candidate key or super key. Hence given relation is not in BCNF.

**Step 4 :** For R being in 3NF for X->Y either i) the X should be candidate key or super key or ii) Y should be prime attribute. (For prime and non prime attributes refer step 1)

- For AB->C or AB->D the AB is a candidate key. Condition for 3NF is satisfied.
- Consider C->A. In this FD the C is not candidate key but A is a prime attribute. Condition for 3NF is satisfied.
- Now consider B->D. In this FD, the B is not candidate key, similarly D is not a prime attribute. Hence condition for 3NF fails over here.

Hence given relation is not in 3NF.

**Step 5 :** For R being in 2NF following condition **should not occur**.

Let X->Y, if X is a proper subset of candidate key and Y is a non prime attribute. This is a case of partial functional dependency.

For relation to be in 2NF there should not be any partial functional dependency.

- For AB->C or AB->D the AB is a **complete candidate key**. Condition for 2NF is satisfied.

- Consider  $C \rightarrow A$ . In this FD the C is not candidate key. Condition for 2NF is satisfied.
  - Now consider  $B \rightarrow D$ . In this FD, the B is a part of candidate key (AB or BC), similarly D is not a prime attribute. That means partial functional dependency occurs here. Hence condition for 2NF fails over here.
- Hence given relation is not in 2NF.

Therefore we can conclude that the given relation R is in 1NF.

**Example 3.12.2** Consider a relation  $R(ABC)$  with following FD  $A \rightarrow B$ ,  $B \rightarrow C$  and  $C \rightarrow A$ .  
What is the normal form of R ?

**Solution :**

**Step 1 :** We will find the candidate key

$$(A)^+ = \{ABC\} = R$$

$$(B)^+ = \{ABC\} = R$$

$$(C)^+ = \{ABC\} = R$$

Hence A, B and C all are candidate keys

**Prime attributes** = {A,B,C}

**Non prime attribute**{}

**Step 2 :** For R being in BCNF for  $X \rightarrow Y$  the X should be candidate key or super key.

From above FDs

- Consider  $A \rightarrow B$  in which A is a candidate key or super key. Condition for BCNF is satisfied.
- Consider  $B \rightarrow C$  in which B is a candidate key or super key. Condition for BCNF is satisfied.
- Consider  $C \rightarrow A$  in which C is a candidate key or super key. Condition for BCNF is satisfied.

This shows that the given relation R is in BCNF.

**Example 3.12.3** Prove that any relational schema with two attributes is in BCNF.

**Solution :** Here, we will consider  $R = \{A, B\}$  i.e. a relational schema with two attributes. Now various possible FDs are  $A \rightarrow B$ ,  $B \rightarrow A$ .

From the above FDs

- Consider  $A \rightarrow B$  in which A is a candidate key or super key. Condition for

- BCNF is satisfied.
- Consider  $B \rightarrow A$  in which B is a candidate key or super key. Condition for BCNF is satisfied.
- Consider both  $A \rightarrow B$  and  $B \rightarrow A$  with both A and B is candidate key or super key. Condition for BCNF is satisfied.
- No FD holds in relation R. In this  $\{A,B\}$  is candidate key or super key. Still condition for BCNF is satisfied.

This shows that any relation R **is in BCNF** with two attributes.



## 2.13 Two Marks Questions with Answers

**Q.1 Explain Entity Relationship model.**

**AU : May-16**

**Ans. :** • The ER data model specifies 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 entities onto a conceptual schema.

**Q.2 Give the limitations of E-R model ? How do you overcome this ?**

**AU : May-07**

**Ans. :** 1) **Loss of information content :** Some information be lost or hidden in ER model

2) **Limited relationship representation :** ER model represents limited relationship as compared to another data models like relational model etc.

3) **No representation of data manipulation :** It is difficult to show data manipulation in ER model.

4) **Popular for high level design :** ER model is very popular for designing high level design.

**Q.3 List the design phases of Entity Relationship model.**

**Ans. :** 1) Requirement Analysis, 2) Conceptual Database Design, 3) Logical Database Design, 4) Schema Refinement, 5) Physical Database Design, 6) Application and Security Design.

**Q.4 What is an entity ?**

**AU : May-14**

**Ans. :** • An entity is an object that exists and is distinguishable from other objects.

- For example - Student named "Poonam" is an entity and can be identified by her name. Entity is represented as a box, in ER model.

**Q.5 What do you mean by derived attributes ?**

**Ans. :** • Derived attributes are the attributes that contain values that are calculated from other attributes.

- To represent derived attribute there is dotted ellipse inside the solid ellipse. For example –Age can be derived from attribute DateOfBirth. In this situation, DateOfBirth might be called Stored Attribute.

**Q.6 What is a weak entity ? Give example.**

**AU : Dec.-16, May-18**

**Ans. :** Refer section 2.3.4

**Q.7 What are the problems caused by redundancy ?**

**AU : Dec.-17**

**Ans. : Problems caused by Redundancy :** Following problems can be caused by redundancy -

- i) **Redundant Storage :** Some information is stored repeatedly.
- ii) **Update Anomalies :** If one copy of such repeated data is updated then inconsistency is created unless all other copies are similarly updated.
- iii) **Insertion Anomalies :** Due to insertion of new record repeated information get added to the relation schema.
- iv) **Deletion Anomalies :** Due to deletion of particular record some other important information associated with the deleted record get deleted and thus we may lose some other important information from the schema.

**Q.8 Define functional dependency.**

**AU : Dec 04,05, May 05,14,15**

**Ans. :** Let P and Q be sets of columns, then : P functionally determines Q, written  $P \rightarrow Q$  if and only if any two rows that are equal on (all the attributes in) P must be equal on (all the attributes in) Q.

In other words, the functional dependency holds if

$$T1.P = T2.P, \text{ then } T1.Q=T2.Q$$

Where notation T1.P projects the tuple T1 onto the attribute in P.

**Q.9 Why certain functional dependencies are called trivial functional dependencies ?**

**AU : May-06,12**

**Ans. :** • A functional dependency  $FD : X \rightarrow Y$  is called trivial if Y is a subset of X. This kind of dependency is called trivial because it can be derived from common sense. If one "side" is a subset of the other, it's considered trivial. The left side is considered the determinant and the right the dependent.

- **For example -**  $\{A,B\} \rightarrow B$  is a trivial functional dependency because B is a subset of A,B. Since  $\{A,B\} \rightarrow B$  includes B, the value of B can be determined. It's a trivial functional dependency because determining B is satisfied by its relationship to A,B

**Q.10 Define normalization.**

**AU : May -14**

**Ans. :** Normalization is the process of reorganizing data in a database so that it meets two basic requirements :

- 1) There is **no redundancy** of data (all data is stored in only one place), and
- 2) **data dependencies** are logical (all related data items are stored together)

**Q.11 State anomalies of 1NF.**

**AU : Dec.-15**

**Ans. :** All the insertion, deletion and update anomalies are in 1NF relation

**Q.12 What is multivalued dependency ?**

**AU : Dec. -06**

**Ans. :** A table is said to have multi-valued dependency, if the following conditions are true,

- 1) For a dependency  $A \twoheadrightarrow B$ , if for a single value of A, **multiple values** of B exists, then the table may have multi-values dependency.
- 2) Also, a table should have **at-least 3 columns** for it to have a multi-valued dependency.
- 3) And, for a relation  $R(A,B,C)$ , if there is a multi-valued **dependency between**, A and B, then B and C should be independent of each other.

**Q.13 Describe BCNF and describe a relation which is in BCNF.**

**AU : Dec. -02**

**Ans. :** Refer section 3.12.

**Q.14 Why 4NF in normal form is more desirable than BCNF ?**

**AU : Dec. -14**

**Ans. :**

- 4NF is more desirable than BCNF because it reduces the repetition of information.
- If we consider a BCNF schema not in 4NF we observe that decomposition into 4NF does not lose information provided that a lossless join decomposition is used, yet redundancy is reduced.

**Q.15 Give an example of a relation schema R and set of dependencies such that R is in BCNF but not in 4NF.**

**AU : May -12**

**Ans. :** Consider relation  $R(A,B,C,D)$  with dependencies

$AB \twoheadrightarrow C$

$ABC \twoheadrightarrow D$

$AC \twoheadrightarrow B$

Here the only key is AB. Thus each functional dependency has superkey on the left. But MVD has non-superkey on its left. So it is not 4NF.

**Q.16 Show that if a relation is in BCNF, then it is also in 3NF.**

**AU : Dec.-12**

**Ans. :**

- Boyce and Codd Normal Form is a **higher version** of the Third Normal form.
- A 3NF table which **does not have multiple overlapping** candidate keys is said to be in BCNF. When the table is in BCNF then it doesn't have partial functional dependency as well as transitive dependency.
- Hence it is true that if relation is in BCNF then it is also in 3NF.

**Q.17 Why it is necessary to decompose a relation ?**

**AU : May-07**

**Ans. :** • Decomposition is the process of breaking down one table into multiple tables.

- The decomposition is used for eliminating redundancy.

**Q.18 Explain atleast two desirable properties of decomposition.**

**AU : May-03,17, Dec.-05**

**Ans. :**

There are two properties associated with decomposition and those are –

- 1) **Loss-less Join or non Loss Decomposition :** When all information found in the original database is preserved after decomposition, we call it as loss less or non loss decomposition.
- 2) **Dependency Preservation :** This is a property in which the constraints on the original table can be maintained by simply enforcing some constraints on each of the smaller relations.

**Q.19 Explain with simple example lossless join decomposition.**

**AU : May-03**

**Ans. :** Refer section 3.10.1.

---

## UNIT - IV

### TRANSACTION MANAGEMENT

#### *Syllabus*

*Transaction Concepts - ACID Properties - Schedules - Serializability - Concurrency Control - Need for Concurrency - Locking Protocols - Two Phase Locking*

#### *Contents*

4.1 Transaction Concepts .....	<b>Dec.-14</b> .....	Marks 4
4.2 ACID Properties .....	<b>May-14, 18</b> .....	Marks 8
4.3 Transaction States .....	<b>Dec.-11, May-14,18</b> .....	Marks 8
4.4 Schedules		
4.5 Serializability .....	<b>Dec.-15, May-15,18</b> .....	Marks 8
4.6 Transaction Isolation and Atomicity		
4.7 Introduction to Concurrency Control		
4.8 Need for Concurrency .....	<b>May-17</b> .....	Marks 13
4.9 Locking Protocols.....	<b>Dec-15,17, May-16</b> .....	Marks 16
4.10 Two Phase Locking .....	<b>May-14,18, Dec.-16</b> .....	Marks
4.11 Two Marks Questions with Answers		

## Part I : Introduction Transactions

### 4.1 Transaction Concepts

AU : Dec.-14, Marks 4

**Definition :** A transaction can be defined as a **group of tasks** that form a single logical unit.

**For example -** Suppose we want to withdraw ` 100 from an account then we will follow following operations :

- 1) Check account balance
- 2) If sufficient balance is present request for withdrawal.
- 3) Get the money
- 4) Calculate Balance = Balance -100
- 5) Update account with new balance.

The above mentioned **four steps** denote **one transaction**.

In a database, each transaction should maintain ACID property to meet the consistency and integrity of the database.

#### University Question

1. Write a short note on – Transaction Concept

AU : Dec.-14, Marks 4

### 4.2 ACID Properties

AU : May-14, Marks 8

#### 1) Atomicity :

- This property states that each transaction must be considered as a **single unit** and **must be completed fully or not completed at all**.
- No transaction in the database is left half completed.
- Database should be in a state either before the transaction execution or after the transaction execution. It **should not be in a state 'executing'**.
- For example - In above mentioned withdrawal of money transaction all the five steps must be completed fully or none of the step is completed. Suppose if transaction gets failed after step 3, then the customer will get the money but the balance will not be updated accordingly. The state of database should be either at before ATM withdrawal (i.e customer without withdrawn money) or after ATM withdrawal (i.e. customer with money and account updated). This will make the system in consistent state.

## 2) Consistency :

- The database must remain in consistent state after performing any transaction.
- For example : In ATM withdrawal operation, the balance must be updated appropriately after performing transaction. Thus the database can be in consistent state.

## 3) Isolation :

- In a database system where **more than one transaction** are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system.
- No transaction will affect the existence of any other transaction.
- **For example** : If a bank manager is checking the account balance of particular customer, then manager should see the balance either before withdrawing the money or after withdrawing the money. This will make sure that each individual transaction is completed and any other dependent transaction will get the consistent data out of it. Any failure to any transaction will not affect other transaction in this case. Hence it makes all the transactions consistent.

## 4) Durability :

- The database should be **strong enough** to handle any **system failure**.
- If there is any set of insert /update, then it should be able to handle and commit to the database.
- If there is any **failure**, the database should be able to **recover** it to the consistent state.
- For example : In ATM withdrawal example, if the system failure happens after Customer getting the money then the system should be strong enough to update Database with his new balance, after system recovers. For that purpose the system has to keep the **log of each transaction and its failure**. So when the system recovers, it should be able to know when a system has failed and if there is any pending transaction, then it should be updated to Database.

## University Questions

1. Explain with an example the properties that must be satisfied by transaction

**AU : May-18, Marks 7**

2. Explain the ACID properties of transaction

**AU : May-14, Marks 8**

### 4.3 Transaction States

AU : Dec.-11, May-14,18, Marks 8

Each transaction has following five states :

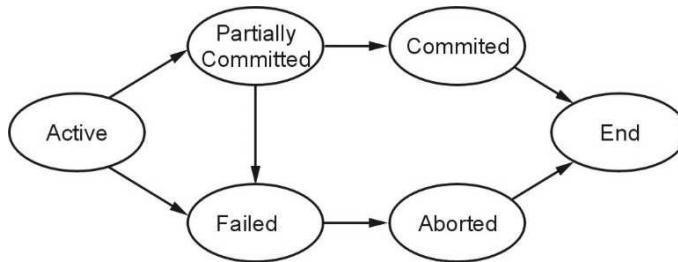


Fig. 4.4.1 Transaction States

- 1) **Active** : This is the first state of transaction. For example : insertion, deletion or updation of record is done here. But data is not saved to database.
- 2) **Partially Committed** : When a transaction executes its final operation, it is said to be in a partially committed state.
- 3) **Failed** : A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- 4) **Aborted** : If a transaction is failed to execute, then the database recovery system will make sure that the database is in its previous consistent state. If not, it brings the database to consistent state by aborting or rolling back the transaction.
- 5) **Committed** : If a transaction executes all its operations successfully, it is said to be committed. This is the last step of a transaction, if it executes without fail.

**Example 4.4.1** Define a transaction. Then discuss the following with relevant examples :

1. A read only transaction
2. A read write transaction
3. An aborted transaction

AU : Dec.-11, Marks 8

**Solution :**

**(1) Read only transaction**

T1
Read(A)
Read(B)
Display(A-B)

**(2) A read write transaction**

T1
Read(A)
A=A+100
Write(A)



(3)

<b>T1</b>	<b>T2</b>	
Read(A)		Assume A=100
A=A+50		A=150
Write(A)		
	Read(A)	A=150
	A=A+100	A=250
RollBack		A=100 (restore back to original value which is before Transaction T1)
	Write(A)	

### University Questions

1. During execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which transaction may pass. Explain why each state transaction may occur ?

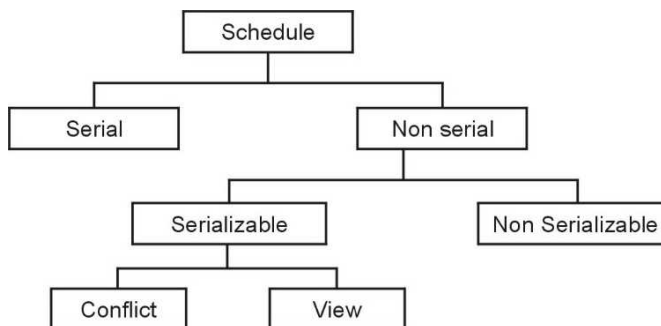
**AU : May-18, Marks 6**

2. With a neat sketch explain the states of transaction.

**AU : May-14, Marks 8**

### 4.4 Schedules

Schedule is an order of multiple transactions executing in concurrent environment. Following figure represents the types of schedules.



**Fig. 4.4.1 : Types of schedule**

**Serial Schedule :** The schedule in which the transactions execute one after the other is called serial schedule. It is consistent in nature. For example : Consider following two transactions  $T_1$  and  $T_2$

T <sub>1</sub>	T <sub>2</sub>
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

All the operations of transaction T<sub>1</sub> on data items A and then B executes and then in transaction T<sub>2</sub> all the operations on data items A and B execute. The **R stands for Read** operation and **W stands for write** operation.

**Non Serial Schedule :** The schedule in which operations present within the transaction are intermixed. This may lead to conflicts in the result or inconsistency in the resultant data. For example-

Consider following two transactions,

T <sub>1</sub>	T <sub>2</sub>
R(A)	
W(A)	
	R(A)
	W(B)
R(A)	
W(B)	
	R(B)
	W(B)

The above transaction is said to be **non serial** which result in inconsistency or conflicts in the data.

## 4.5 Serializability

AU : Dec.-15, May-15, 18, Marks 8

- When multiple transactions run concurrently, then it may lead to inconsistency of data (i.e. change in the resultant value of data from different transactions).
- Serializability is a concept that helps to identify which non serial schedule and find the transaction equivalent to serial schedule.

- For example :

T <sub>1</sub>	A	B	T <sub>2</sub>
Initial Value	100	100	
A=A-10			
W(A)			
B=B+10			
W(B)			
	90	110	
			A=A-10
			W(A)
	80	110	

- In above transactions initially T<sub>1</sub> will read the values from database as A=100, B=100 and modify the values of A and B. But transaction T<sub>2</sub> will read the modified value i.e. 90 and will modify it to 80 and perform write operation. Thus at the end of transaction T<sub>1</sub> value of A will be 90 but at end of transaction T<sub>2</sub> value of A will be 80. Thus conflicts or inconsistency occurs here. This sequence can be converted to a sequence which may give us consistent result. This process is called **serializability**.

### Difference between Serial Schedule and Serializable Schedule

Serial Schedule	Serializable Schedule
<b>No concurrency</b> is allowed in serial schedule.	<b>Concurrency</b> is allowed in serializable schedule.
In serial schedule, if there are two transactions executing at the same time and no interleaving of operations is permitted, then following can be the possibilities of execution – (i) Execute all the operations of transactions T1 in a sequence and then execute all the operations of transactions T2 in a sequence. (ii) Execute all the operations of transactions T2 in a sequence and then execute all the operations of transactions T1 in a sequence.	In serializable schedule, if there are two transactions executing at the same time and interleaving of operations is allowed there can be different possible orders of executing an individual operation of the transactions.

Example of Serial Schedule		Example of Serializable Schedule	
T1	T2	T1	T2
Read(A)		Read(A)	
A=A-50		A=A-50	
Write(A)		Write(A)	
Read(B)			Read(B)
B=B+100			B=B+100
Write(B)			Write(B)
	Read(A)	Read(B)	
	A=A+10	Write(B)	
	Write(A)		

- There are **two types of serializabilities** : conflict serializability and view serializability

#### **4.5.1 Conflict Serializability**

**Definition** : Suppose  $T_1$  and  $T_2$  are two transactions and  $I_1$  and  $I_2$  are the instructions in  $T_1$  and  $T_2$  respectively. Then these two transactions are said to be conflict Serializable, if both the instruction access the data item  $d$ , and at least one of the instruction is write operation.

**What is conflict ?**: In the definition **three conditions** are specified for a conflict in conflict serializability –

- 1) There should be **different transactions**
  - 2) The **operations** must be performed on **same data** items
  - 3) **One of the operation** must be the **Write(W)** operation
- We can test a given schedule for conflict serializability by constructing a **precedence graph** for the schedule, and by searching for absence of cycles in the graph.
  - Precedence graph is a directed graph, consisting of  $G=(V,E)$  where  $V$  is set of vertices and  $E$  is set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges  $T_i \rightarrow T_j$  for which one of three conditions holds :
    1.  $T_i$  executes write(Q) before  $T_j$  executes read(Q).
    2.  $T_i$  executes read(Q) before  $T_j$  executes write(Q).

3.  $T_i$  executes write(Q) before  $T_j$  executes write(Q).

- A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

### Testing for serializability

Following method is used for testing the serializability : To test the conflict serializability we can draw a graph  $G=(V,E)$  where  $V$  = vertices which represent the number of transactions.  $E$  = edges for conflicting pairs.

**Step 1 :** Create a node for each transaction.

**Step 2 :** Find the conflicting pairs(RW, WR, WW) on the same variable(or data item) by different transactions.

**Step 3 :** Draw edge for the given schedule. Consider following cases

1.  $T_i$  executes write(Q) before  $T_j$  executes read(Q), then draw edge from  $T_i$  to  $T_j$ .
2.  $T_i$  executes read(Q) before  $T_j$  executes write(Q) , then draw edge from  $T_i$  to  $T_j$
3.  $T_i$  executes write(Q) before  $T_j$  executes write(Q), , then draw edge from  $T_i$  to  $T_j$

**Step 4 :** Now, if precedence graph is cyclic then it is a non conflict serializable schedule and if the precedence graph is acyclic then it is conflict serializable schedule.

**Example 3.5.1** Consider the following two transactions and schedule (time goes from top to bottom). Is this schedule conflict-serializable ? Explain why or why not.

$T_1$	$T_2$
R(A)	
W(A)	
	R(A)
	R(B)
R(B)	
W(B)	

**Solution :**

**Step 1 :** To check whether the schedule is conflict serializable or not we will check from **top to bottom**. Thus we will start reading from top to bottom as

$T_1: R(A) \rightarrow T_1: W(A) \rightarrow T_2: R(A) \rightarrow T_2: R(B) \rightarrow T_1: R(B) \rightarrow T_1: W(B)$

**Step 2 :** We will find **conflicting operations**. Two operations are called as conflicting operations if all the following conditions hold true for them-

- i) Both the **operations belong to different transactions**.
- ii) Both the operations are on **same data item**.
- iii) **At least one** of the two operations is a **write operation**

From above given example in the top to bottom scanning we find the conflict as  $T_1:W(A) \rightarrow T_2:R(A)$ .

- i) Here note that there are two different transactions  $T_1$  and  $T_2$ ,
- ii) Both work on same data item i.e. A and
- iii) One of the operation is write operation

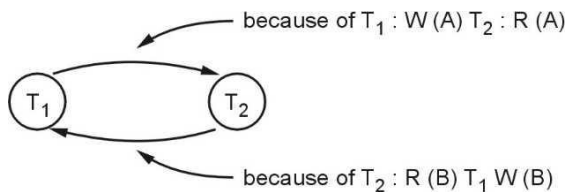
**Step 3 :** We will build a precedence graph by drawing one node from each transaction. In above given scenario as there are two transactions, there will be two nodes namely  $T_1$  and  $T_2$



**Step 4 :** Draw the edge between conflicting transactions. For example in above given scenario, the conflict occurs while moving from  $T_1:W(A)$  to  $T_2:R(A)$ . Hence edge must be from  $T_1$  to  $T_2$ .



**Step 5 :** Repeat the step 4 while reading from top to bottom. Finally the **precedence graph** will be as follows



**Fig. 4.5.1 : Precedence graph**

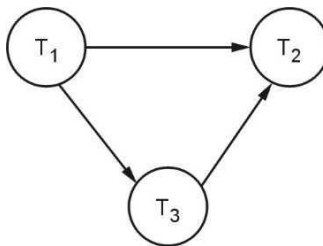
**Step 6 :** Check if **any cycle** exists in the graph. Cycle is a path using which we can start from one node and reach to the same node. If the is cycle found then schedule is not conflict serializable. In the step 5 we get **a graph with cycle**, that means given schedule is **not conflict serializable**.

**Example 4.5.2** Check whether following schedule is conflict serializable or not. If it is not conflict serializable then find the serializability order.

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>
R(A)		
	R(B)	
		R(B)
	W(B)	
W(A)		
		W(A)
	R(A)	
	W(A)	

**Solution :**

**Step 1 :** We will read from top to bottom, and build a precedence graph for conflicting entries :



**Fig. 4.5.2 Precedence graph**

**Step 2 :** As there is **no cycle** in the precedence graph, the given sequence is **conflict serializable**. Hence we can convert this non serial schedule to serial schedule. For that purpose we will follow these steps to find the serializable order.

**Step 3 :** A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

**Step 4 :** Find the vertex which has no incoming edge which is T<sub>1</sub>. Finally find the vertex having no outgoing edge which is T<sub>2</sub>. So in between them is T<sub>3</sub>. Hence the order will be T<sub>1</sub> - T<sub>3</sub> - T<sub>2</sub>

#### **4.5.2 View Serializability**

If a given schedule is found to be view equivalent to some serial schedule, then it is called as a view serializable schedule.

**View Equivalent Schedule :** Consider two schedules S<sub>1</sub> and S<sub>2</sub> consisting of transactions

$T_1$  and  $T_2$  respectively, then schedules  $S_1$  and  $S_2$  are said to be view equivalent schedule if it satisfies following three conditions :

- If transaction  $T_i$  reads a data item A from the database initially in schedule  $S_1$ , then in schedule  $S_2$  also,  $T_i$  must perform the initial read of the data item X from the database. This is same for all the data items. In other words - the initial reads must be same for all data items.
- If data item A has been updated at last by transaction  $T_i$  in schedule  $S_1$ , then in schedule  $S_2$  also, the data item A must be updated at last by transaction  $T_i$ .
- If transaction  $T_i$  reads a data item that has been updated by the transaction  $T_j$  in schedule  $S_1$ , then in schedule  $S_2$  also, transaction  $T_i$  must read the same data item that has been updated by transaction  $T_j$ . In other words the Write-Read sequence must be same.

### Steps to check whether the given schedule is view serializable or not

**Step 1 :** If the schedule is conflict serializable then it is surely view serializable because conflict serializability is a restricted form of view serializability.

**Step 2 :** If it is not conflict serializable schedule then check whether there exist any blind write operation. The blind write operation is a write operation without reading a value. If there does not exist any blind write then that means the given schedule is not view serializable. In other words if a blind write exists then that means schedule may or may not be view conflict.

**Step 3 :** Find the view equivalence schedule

**Example 4.5.3** Consider the following schedules for checking if these are view serializable or not.

$T_1$	$T_2$	$T_3$
		W(C)
	R(A)	
	W(B)	
R(C)		
		W(B)
W(B)		

**Solution :** i) The initial read operation is performed by  $T_2$  on data item A or by  $T_1$  on data item C. Hence we will begin with  $T_2$  or  $T_1$ . We will choose  $T_2$  at the beginning.



ii) The final write is performed by  $T_1$  on the same data item B. Hence  $T_1$  will be at the last position.

iii) The data item C is written by  $T_3$  and then it is read by  $T_1$ . Hence  $T_3$  should appear before  $T_1$ .

Thus we get the order of schedule of view serializability as  $T_2 - T_1 - T_3$

**Example 4.5.4** Consider following two transactions :

$T_1$  : read(A)

read(B)

if A=0 then B:=B+1;

write(B)

$T_2$  : read(B);

read(A);

if B=0 then A:=A+1;

write(A)

Let consistency requirement be  $A=0 \vee B=0$  with  $A=B=0$  the initial values.

- 1) Show that every serial execution involving these two transactions preserves the consistency of the Database ?
- 2) Show a concurrent execution of  $T_1$  and  $T_2$  that produces a non serializable schedule ?
- 3) Is there a concurrent execution of  $T_1$  and  $T_2$  that produces a serializable schedule ?

**Solution : 1)** There are two possible executions:  $T_1 \rightarrow T_2$  or  $T_2 \rightarrow T_1$

Consider case  $T_1 \rightarrow T_2$  then

A	B
0	0
0	1
0	1

$A \vee B = A$  OR  $B = F \vee T = T$ . This means consistency is met.

Consider case  $T_2 \rightarrow T_1$  then

A	B
0	0
1	0
1	0

$A \vee B = A$  OR  $B = F \vee T = T$ . This means consistency is met.

(2) The concurrent execution means interleaving of transactions  $T_1$  and  $T_2$ . It can be

$T_1$	$T_2$
R(A)	
	R(B)
	R(A)
R(B) If A=0 then B=B+1	If B=0 then A=A+1 W(A)
W(B)	

This is a non-serializable schedule.

(3) There is no concurrent execution resulting in a serializable schedule.

**Example 4.5.5** Test serializability of the following schedule :

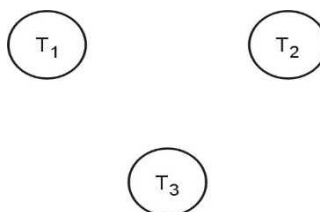
i)  $r_1(x); r_3(x); w_1(x); r_2(x); w_3(x)$  ii)  $r_3(x); r_2(x); w_3(x); r_1(x); w_1(x)$

**Solution :** i)  $r_1(x); r_3(x); w_1(x); r_2(x); w_3(x)$

The  $r_1$  represents the read operation of transaction  $T_1$ ,  $w_3$  represents the write operation on transaction  $T_3$  and so on. Hence from given sequence the schedule for three transactions can be represented as follows :

$T_1$	$T_2$	$T_3$
$r_1(x)$		
		$r_3(x)$
$w_1(x)$		
	$r_2(x)$	
		$w_3(x)$

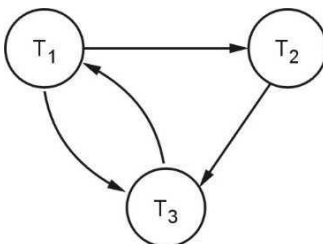
**Step 1 :** We will use the precedence graph method to check the serializability. As there are three transactions, three nodes are created for each transaction.



**Fig. 4.5.3 Nodes**

**Step 2 :** We will read from top to bottom. Initially we read  $r_1(x)$  and keep on moving bottom in search of write operation. Here all the transactions work on same data item i.e.  $x$ . Now we get a write operation in  $T_3$  as  $w_3(x)$ . Hence the dependency is from  $T_1$  to  $T_3$ . Therefore we draw edge from  $T_1$  to  $T_3$ .

Similarly, for  $r_3(x)$  we get  $w_1(x)$  pair. Hence there will be edge from  $T_3$  to  $T_1$ . Continuing in this fashion we get the precedence graph as



**Fig. 4.5.4 Precedence Graph**

**Step 3 :** As cycle exists in the above precedence graph, we conclude that it is **not serializable**.

ii)  $r_3(x);r_2(x);w_3(x);r_1(x);w_1(x)$

From the given sequence the schedule can be represented as follows :

$T_1$	$T_2$	$T_3$
		$r_3(x)$
	$r_2(x)$	
		$w_3(x)$
$r_1(x)$		
$w_1(x)$		

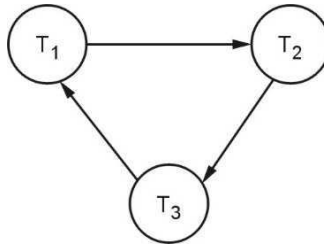
**Step 1 :** Read the schedule from top to bottom for pair of operations. For  $r_3(x)$  we get  $w_1(x)$  pair. Hence edge exists from  $T_3$  to  $T_1$  in precedence graph.

There is a pair from  $r_2(x) : w_3(x)$ . Hence edge exists from  $T_2$  to  $T_3$ .

There is a pair from  $r_1(x) : w_1(x)$ . Hence edge exists from  $T_1$  to  $T_3$ .

There is a pair from  $w_3(x) : r_1(x)$ . Hence edge exists from  $T_3$  to  $T_1$ .

**Step 2 :** The precedence graph will then be as follows –

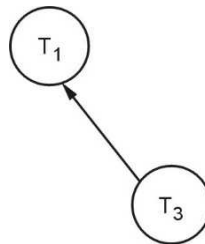


**Fig. 4.5.5 Precedence graph**

**Step 3 :** As there is no cycle in the above graph, the given schedule is **serializable**.

**Step 4 :** The serializability order for consistent schedule will be obtained by applying topological sorting on above drawn precedence graph. This can be achieved as follows,

**Sub-Step 1 :** Find the node having no incoming edge. We obtain  $T_2$  is such a node. Hence  $T_2$  is at the beginning of the serializability sequence. Now delete  $T_2$ . The Graph will be



**Sub-Step 2 :** Repeat sub-Step 1, We obtain  $T_3$  and  $T_1$  nodes as a sequence.

Thus we obtain the sequence of transactions as  $T_2, T_3$  and  $T_1$ . Hence the serializability order is

$$r_2(x);r_3(x);w_3(x);r_1(x);w_1(x)$$

**Example 3.5.6** Consider the following schedules. The actions are listed in the order they are scheduled, and prefixed with the transaction name.

$S1 : T1 : R(X), T2 : R(X), T1 : W(Y), T2 : W(Y) T1 : R(Y), T2 : R(Y)$

$S2 : T3 : W(X), T1 : R(X), T1 : W(Y), T2 : R(Z), T2 : W(Z) T3 : R(Z)$

For each of the schedules, answer the following questions :

i) What is the precedence graph for the schedule ?

ii) Is the schedule conflict-serializable ? If so, what are all the conflict equivalent serial schedules ?

iii) Is the schedule view-serializable ? If so, what are all the view equivalent serial schedules ?

**AU : May-15, Marks 2 + 7 + 7**

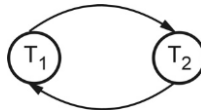
**Solution : i)** We will find **conflicting operations**. Two operations are called as conflicting operations if all the following conditions hold true for them-

- Both the **operations belong to different transactions**.
- Both the operations are on **same data item**.
- At least one of the two operations is a write operation

**For S1:** From above given example in the top to bottom scanning we find the conflict as

- **T1 : W(Y), T2 : W(Y) and**
- **T2 : W(Y), T1 : R(Y)**

Hence we will build the precedence graph. Draw the edge between conflicting transactions. For example in above given scenario, the conflict occurs while moving from  $T_1:W(Y)$  to  $T_2:W(Y)$ . Hence edge must be from  $T_1$  to  $T_2$ . Similarly for second conflict, there will be the edge from  $T_2$  to  $T_1$

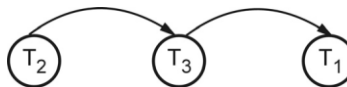


**Fig. 4.5.6 Precedence graph for S1**

**For S2:** The conflicts are

- **T3 : W(X), T1 : R(X)**
- **T2 : W(Z) T3 : R(Z)**

Hence the precedence graph is as follows -



**Fig. 4.5.7 Precedence graph for S2**

**(ii)**

- S1 is **not conflict-serializable** since the dependency graph has a cycle.
- S2 is conflict-serializable as the dependency graph is acyclic. The order **T2-T3-T1** is the only equivalent **serial order**.

**(iii)**

- S1 is **not view serializable**.
- S2 is trivially view-serializable as it is conflict serializable. The **only serial order** allowed is

**T2-T3-T1.**

## University Questions

1. Explain Conflict serializability and view serializability.

AU : May-18, Marks 6, Dec.-15, Marks 8

### 4.6 Transaction Isolation and Atomicity

- The serializable schedule can be made consistent by applying conflict serializability or view serializability.
- The serializable order makes a transaction isolation. But during the execution of concurrent transactions in a given schedule, some of the transaction may get failed(may be due to hardware or software failure)
- If the transaction gets failed we need to undo the effect of this transaction to ensure the atomicity property of transaction. This makes the schedule acceptable even after the failure.
- The schedule can be made acceptable by two techniques namely – Recoverable Schedule and Cascadeless schedule.

#### 4.6.1 Recoverable Schedule

**Definition :** A recoverable schedule is one where, for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the commit operation of  $T_j$

For example : Consider following schedule, consider  $A=100$

$T_1$	$T_2$
R(A)	
$A=A+50$	
W(A)	
	R(A)
	$A=A-20$
	W(A)
	Commit
some transaction...	
Commit	

Failure

- The above schedule is inconsistent if failure occurs after the commit of T<sub>2</sub>.
- It is because T<sub>2</sub> is **dependable transaction** on T<sub>1</sub>. A transaction is said to be dependable if it contains a **dirty read**.
- The dirty read is a situation in which one transaction reads the data immediately after the write operation of previous transaction

T <sub>1</sub>	T <sub>2</sub>
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A-20
	W(A)
	Commit
Commit	

Dirty read

- Now if the dependable transaction i.e. T<sub>2</sub> is committed first and then failure occurs then if the transaction T<sub>1</sub> makes any changes then those changes will not be known to the T<sub>2</sub>. This leads to non recoverable state of the schedule.
- To make the schedule recoverable we will apply the rule that - commit the independent transaction before any dependable transaction.
- In above example independent transaction is T<sub>1</sub>, hence we must commit it before the dependable transaction i.e. T<sub>2</sub>.
- The recoverable schedule will then be -

T <sub>1</sub>	T <sub>2</sub>
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A-20
	W(A)
Commit	
	Commit

### 4.6.2 Cascadeless Schedule

**Definition :** If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written that data item is committed or aborted, then such a schedule is known as a cascadeless schedule.

The cascadeless schedule allows only **committed Read** operation. For example :

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>
R(A)		
A=A+50		
W(A)		
Commit		
	R(A)	
	A=A-20	
	W(A)	
	Commit	
		R(A)
		W(A)

In above schedule at any point if the failure occurs due to commit operation before every Read operation of each transaction, the schedule becomes recoverable and atomicity can be maintained.

## Part II : Concurrency Control

### 4.7 Introduction to Concurrency Control

- One of the fundamental properties of a transaction is **isolation**.
- When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.



- A database can have multiple transactions running at the same time. This is called **concurrency**.
- To preserve the isolation property, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called **concurrency control schemes**.
- **Definition of concurrency control** : A mechanism which ensures that simultaneous execution of more than one transactions does not lead to any database inconsistencies is called **concurrency control mechanism**.
- The concurrency control can be achieved with the help of various protocols such as - lock based protocol, Deadlock handling, Multiple Granularity, Timestamp based protocol, and validation based protocols.

#### 4.8 Need for Concurrency

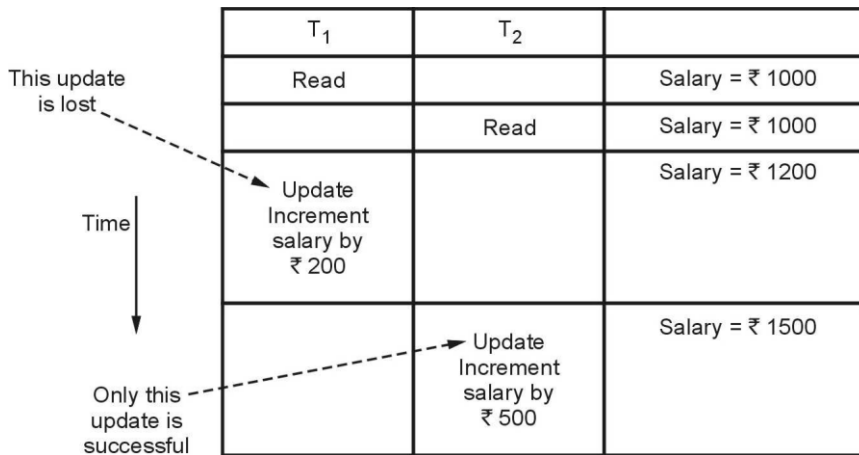
**AU : May-17, Marks 13**

- Following are the purposes of concurrency control –
  - **To ensure isolation**
  - **To resolve read-write or write-write conflicts**
  - **To preserve consistency of database**
- Concurrent execution of transactions over shared database creates several data integrity and consistency problems – these are

**(1) Lost Update Problem** : This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

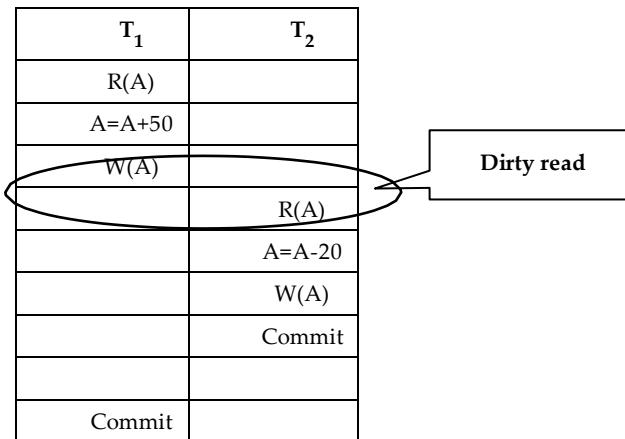
For example – Consider following transactions

- (1) Salary of Employee is read during transaction T1.
- (2) Salary of Employee is read by another transaction T2.
- (3) During transaction T1, the salary is incremented by ` 200
- (4) During transaction T2, the salary is incremented by ` 500



The result of the above sequence is that the update made by transaction T1 is completely lost. Therefore this problem is called as lost update problem.

**(2) Dirty Read or Uncommitted Read Problem :** The dirty read is a situation in which one transaction reads the data immediately after the write operation of previous transaction



For example – Consider following transactions -

Assume initially salary is = ` 1000

	T <sub>1</sub>	T <sub>1</sub>	
Time ↓	...	...	Salary = ₹ 1000
t <sub>1</sub>		Update Salary = Salary + 200	Salary = ₹ 1200
t <sub>2</sub>		Read	Salary = ₹ 1200
t <sub>3</sub>		Rollback	Salary = ₹ 1000

Dirty Read

- (1) At the time t<sub>1</sub>, the transaction T<sub>2</sub> updates the salary to `1200
- (2) This salary is read at time t<sub>2</sub> by transaction T<sub>1</sub>. Obviously it is ` 1200
- (3) But at the time t<sub>3</sub>, the transaction T<sub>2</sub> performs Rollback by undoing the changes made by T<sub>1</sub> and T<sub>2</sub> at time t<sub>1</sub> and t<sub>2</sub>.
- (4) Thus the salary again becomes = ` 1000. This situation leads to **Dirty Read or Uncommitted Read** because here the read made at time t<sub>2</sub>(**immediately after update of another transaction**) becomes a dirty read.

### (3) Non-repeatable read Problem

This problem is also known as **inconsistent analysis problem**. This problem occurs when a particular transaction sees two different values for the same row within its lifetime. For example –

	T <sub>1</sub>	T <sub>2</sub>	
Time ↓	t <sub>1</sub>	Read	Salary = ₹ 1000
t <sub>2</sub>		Update salary from ₹ 1000 to ₹ 1200	Salary = ₹ 1200
t <sub>3</sub>		Commit	
t <sub>4</sub>	Read		Salary = ₹ 1200

- (1) At time t<sub>1</sub>, the transaction T<sub>1</sub> reads the salary as ` 1000
- (2) At time t<sub>2</sub> the transaction T<sub>2</sub> reads the same salary as ` 1000 and updates it to `1200
- (3) Then at time t<sub>3</sub>, the transaction T<sub>2</sub> gets committed.
- (4) Now when the transaction T<sub>1</sub> reads the same salary at time t<sub>4</sub>, it gets different value than what it had read at time t<sub>1</sub>. Now, transaction T<sub>1</sub> cannot repeat its reading operation. Thus inconsistent values are obtained.

Hence the name of this problem is non-repeatable read or inconsistent analysis problem.

#### (4) Phantom read Problem

The phantom read problem is a special case of non repeatable read problem.

This is a problem in which one of the transaction makes the changes in the database system and due to these changes another transaction can not read the data item which it has read just recently. For example –

	T <sub>1</sub>	T <sub>2</sub>	
t <sub>1</sub>	Read		Salary = ₹ 1000
t <sub>2</sub>		Read	Salary = ₹ 1000
t <sub>3</sub>	Delete salary		No salary
t <sub>4</sub>		Read	"Can not find salary"

- (1) At time t<sub>1</sub>, the transaction T<sub>1</sub> reads the value of salary as ` 1000
- (2) At time t<sub>2</sub>, the transaction T<sub>2</sub> reads the value of the same salary as ` 1000
- (3) At time t<sub>3</sub>, the transaction T<sub>1</sub> deletes the variable salary.
- (4) Now at time t<sub>4</sub>, when T<sub>2</sub> again reads the salary it gets error. Now transaction T<sub>2</sub> can not identify the reason why it is not getting the salary value which is read just few time back.

This problem occurs due to changes in the database and is called phantom read problem.

#### University Question

1. Discuss the violations caused by each of the following: dirty read, non repeatable read and phantoms with suitable example

**AU : May-17, Marks 13**

## 4.9 Locking Protocols

**AU : May-16, Dec.-15, 17, Marks 16**

### 4.9.1 Why Do we Need Locks ?

- One of the method to ensure the **isolation** property in transactions is to require that data items be accessed in a mutually exclusive manner. That means, 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**.
- Thus the lock on the operation is required to ensure the isolation of transaction.

## 4.9.2 Simple Lock Based Protocol

- **Concept of Protocol** : The lock based protocol is a mechanism in which there is exclusive use of **locks** on the data item for current transaction.
- **Types of Locks** : There are two types of locks used -

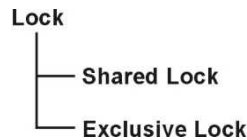


Fig. 4.9.1 Types of locks

- Shared Lock** : The shared lock is used for reading data items only. It is denoted by **Lock-S**. This is also called as **read lock**.
  - Exclusive Lock** : The exclusive lock is used for both read and write operations. It is denoted as **Lock-X**. This is also called as **write lock**.
- The **compatibility matrix** is used while working on set of locks. The **concurrency control manager** checks the compatibility matrix before granting the lock. If the two modes of transactions are compatible to each other then only the lock will be granted.
  - In a set of locks may consists of shared or exclusive locks. Following matrix represents the compatibility between modes of locks.

	S	X
S	T	F
X	F	F

Fig. 4.9.2 Compatibility matrix for locks

Here T stands for True and F stands for False. If the control manager get the compatibility mode as True then it grant the lock otherwise the lock will be denied.

- **For example** : If the transaction  $T_1$  is holding a shared lock in data item A, then the control manager can grant the shared lock to transaction  $T_2$  as compatibility is True. But it cannot grant the exclusive lock as the compatibility is false. In simple words if transaction  $T_1$  is reading a data item A then same data item A can be read by another transaction  $T_2$  but cannot be written by another transaction.
- Similarly if an exclusive lock (i.e. lock for read and write operations) is hold on the data item in some transaction then no other transaction can acquire Shared or exclusive lock as the compatibility function denotes F. That means of some

transaction is writing a data item A then another transaction can not read or write that data item A.

Hence the **rule of thumb** is

- i) **Any number** of transactions can hold **shared lock** on an item.
- ii) But **exclusive lock** can be hold by **only one transaction**.

- **Example of a schedule denoting shared and exclusive locks** : Consider following schedule in which initially  $A=100$ . We deduct 50 from A in  $T_1$  transaction and Read the data item A in transaction  $T_2$ . The scenario can be represented with the help of locks and concurrency control manager as follows :

	$T_1$	$T_2$	Concurrency control manager
Exclusive Lock	Lock-X(A)		Grant X(A,T1) because in T1 there is write operation.
	R(A)		
	$A=A-50$		
	W(A)		
	Unlock(A)		
Shared Lock		Lock-S(A)	Grant S(A,T2) because in T2 there is Read operation
		R(A)	
		Unlock(A)	

### University Questions

1. State and explain the lock based concurrency control with suitable example

**AU : Dec-17, Marks 13, May-16, Marks 16**

2. What is Concurrency control ? How is implemented in DBMS ? Illustrate with suitable example.

**AU : Dec-15, Marks 8**

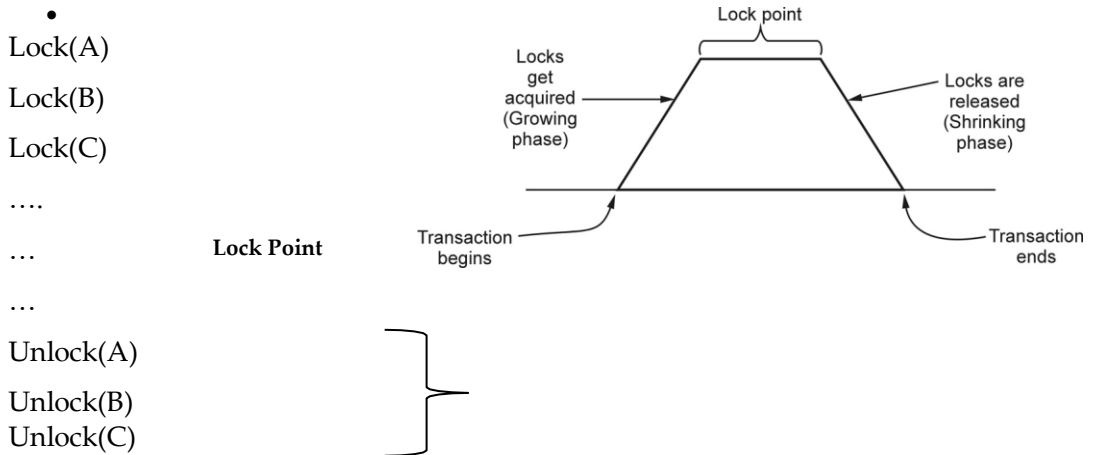
### 4.10 Two Phase Locking

**AU : May-14, 18, Dec.-16, Marks 7**

- The two phase locking is a protocol in which there are two phases :
  - i) **Growing phase (Locking phase)** : It is a phase in which the transaction may obtain locks but does not release any lock.

ii) **Shrinking phase (Unlocking phase)** : It is a phase in which the transaction may release the locks but does not obtain any new lock.

- **Lock Point** : The **last lock position** or **first unlock position** is called lock point. For example



For example –  
Consider following transactions

T1	T2
Lock-X(A)	Lock-S(B)
Read(A)	Read(B)
A=A-50	Unlock-S(B)
Write(A)	
Lock-X(B)	
Unlock-X(A)	
B=B+100	Lock-S(A)
Write(B)	Read(A)
Unlock-X(B)	Unlock-S(A)

The important rule for being a two phase locking is - **All Lock operations precede all the unlock operations.**

In above transactions T1 is in two phase locking mode but transaction T2 is not in two phase locking. Because in T2, the Shared lock is acquired by data item B, then data item B is read and then the lock is released. Again the lock is acquired by data item A , then the data item A is read and the lock is then released. Thus we get lock-unlock-lock-unlock sequence. Clearly this is not possible in two phase locking.

**Example 4.10.1** Prove that two phase locking guarantees serializability.

**Solution:**

- Serializability is mainly an issue of handling write operation. Because any inconsistency may only be created by **write** operation.
- Multiple reads on a database item can happen parallelly.
- 2-Phase locking protocol restricts this unwanted read/write by applying **exclusive lock**.
- Moreover, when there is an **exclusive lock** on an item it will **only be released in shrinking phase**. Due to this restriction there is no chance of getting any inconsistent state.

The serializability using two phase locking can be understood with the help of following example

Consider two transactions

T <sub>1</sub>	T <sub>2</sub>
R(A)	
	R(A)
R(B)	
W(B)	

**Step 1 :** Now we will **apply two phase locking**. That means we will **apply locks in growing** and **shrinking** phase

T <sub>1</sub>	T <sub>2</sub>
Lock-S(A)	
R(A)	
	Lock-S(A)
	R(A)
Lock-X(B)	
R(B)	
W(B)	
Unlock-X(B)	
	Unlock-S(A)



Note that above schedule is serializable as it prevents interference between two transactions.

The serializability order can be obtained based on the **lock point**. The lock point is either last lock operation position or first unlock position in the transaction.

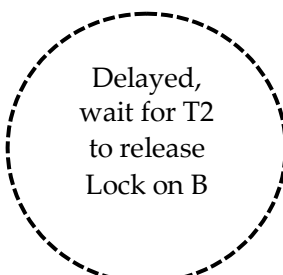
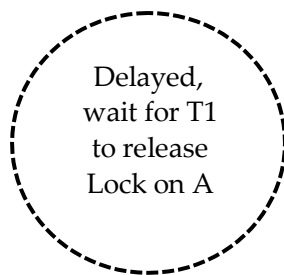
The last lock position is in  $T_1$ , then it is in  $T_2$ . Hence the serializability will be  $T_1 \rightarrow T_2$  based on lock points. Hence The **serializability sequence** can be **R1(A);R2(A);R1(B);W1(B)**

### Limitations of Two Phase Locking Protocol

The two phase locking protocol leads to two problems – deadlock and cascading roll back.

**(1) Deadlock :** The deadlock problem can not be solved by two phase locking. Deadlock is a situation in which when two or more transactions have got a lock and waiting for another locks currently held by one of the other transactions.

For example

T1	T2
Lock-X(A)	Lock-X(B)
Read(A)	Read(B)
A=A-50	B=B+100
Write(A)	Write(B)
 <p>Delayed, wait for T2 to release Lock on B</p>	 <p>Delayed, wait for T1 to release Lock on A</p>

**(2) Cascading Rollback :** Cascading rollback is a situation in which a single transaction failure leads to a series of transaction rollback. For example –

T1	T2	T3
Read(A)		
Read(B)		
C=A+B		
Write(C)		
	Read(C)	
	Write(C)	
		Read(C)

When T1 writes value of C then only T2 can read it. And when T2 writes the value of C then only transaction T3 can read it. But if the transaction T1 gets failed then automatically transactions T2 and T3 gets failed.

The simple two phase locking does not solve the cascading rollback problem. To solve the problem of cascading Rollback two types of two phase locking mechanisms can be used.

#### 4.10.1 Types of Two Phase Locking

**(1) Strict Two Phase Locking :** The strict 2PL protocol is a basic two phase protocol but **all the exclusive mode locks be held until the transaction commits**. That means in other words all the **exclusive locks** are **unlocked** only after the **transaction is committed**. That also means that if T<sub>1</sub> has exclusive lock, then T<sub>1</sub> will release the exclusive lock only after commit operation, then only other transaction is allowed to read or write. For example - Consider two transactions

T <sub>1</sub>	T <sub>2</sub>
W(A)	
	R(A)

If we apply the locks then

T <sub>1</sub>	T <sub>2</sub>
Lock-X(A)	
W(A)	
<b>Commit</b>	
Unlock(A)	
	Lock-S(A)
	R(A)
	Unlock-S(A)

Thus only after commit operation in  $T_1$ , we can unlock the exclusive lock. This ensures the strict serializability.

Thus compared to basic two phase locking protocol, the advantage of strict 2PL protocol is it ensures strict serializability.

**(2) Rigorous Two Phase Locking :** This is stricter two phase locking protocol. Here all locks are to be held until the transaction commits. The transactions can be serialized in the order in which they commit.

example - Consider transactions

<b>T<sub>1</sub></b>
R(A)
R(B)
W(B)

If we apply the locks then

<b>T<sub>1</sub></b>
Lock-S(A)
R(A)
Lock-X(B)
R(B)
W(B)
<b>Commit</b>
Unlock(A)
Unlock(B)

Thus the above transaction uses rigorous two phase locking mechanism

**Example 4.10.2** Consider the following two transactions :

*T1:read(A)*

*Read(B);*

*If A=0 then B=B+1;*

*Write(B)*

*T2:read(B); read(A)*

*If B=0 then A=A+1*

*Write(A)*

*Add lock and unlock instructions to transactions T1 and T2, so that they observe two phase locking protocol. Can the execution of these transactions result in deadlock ?*

**AU : Dec.-16, Marks 6**

**Solution :**

T1	T2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Read(B)	Read(A)
if A=0 then B=B+1	if B=0 then A=A+1
Write(B)	Write(A)
Unlock(A)	Unlock(B)
Commit	Commit
Unlock(B)	Unlock(A)

This is lock-unlock instruction sequence help to satisfy the requirements for strict two phase locking for the given transactions.

The execution of these transactions result in deadlock. Consider following partial execution scenario which leads to deadlock.

T1	T2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Now it will wait for T2 to release exclusive lock on A	Now it will wait for T1 to release exclusive lock on B

#### **4.10.2 Lock Conversion**

Lock conversion is a mechanism in two phase locking mechanism - which allows conversion of shared lock to exclusive lock or exclusive lock to shared lock.

**Method of Conversion :**

**First Phase :**

- can acquire a lock-S on item
- can acquire a lock-X on item
- can convert a lock-S to a lock-X (**upgrade**)

**Second Phase :**

- can release a lock-S
- can release a lock-X
- can convert a lock-X to a lock-S (**downgrade**)

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

For example – Consider following two transactions –

T <sub>1</sub>	T <sub>2</sub>
R(A)	R(A)
R(B)	R(B)
...	
R(C)	
...	
W(A)	

Here if we start applying locks, then we must apply the **exclusive lock** on data item A, because we have to read as well as write on data item A. Another transaction T2 does not get shared lock on A until transaction T1 performs write operation on A. Since transaction T1 needs exclusive lock only at the end when it performs write operation on A, it is better if T1 could initially lock A in shared mode and then later change it to exclusive mode lock when it performs write operation. In such situation, the lock conversion mechanism becomes useful.

When we convert the shared mode lock to exclusive mode then it is called **upgrading** and when we convert exclusive mode lock to shared mode then it is called **downgrading**.

Also note that upgrading takes place only in **growing phase** and downgrading takes place only in shrinking phase. Thus we can refine above transactions using lock conversion mechanism as follows –

T <sub>1</sub>	T <sub>2</sub>
<b>Lock-S(A)</b>	
R(A)	
	<b>Lock-S(A)</b>
	R(A)
<b>Lock-S(B)</b>	
R(B)	
	<b>Lock-S(B)</b>
	R(B)
	Unlock(A)
	Unlock(B)
...	

Lock-S(C)	
R(C)	
...	
<b>Upgrade(A)</b>	
W(A)	
Unlock(A)	
Unlock(B)	
Unlock(C)	

### University Questions

1. What is concurrency control ? Explain two phase locking protocol with an example

**AU : May-18, Marks 7**

2. Illustrate two phase locking protocol with an example.

**AU : May-14, Dec.-16, Marks 6**

### University Question

1. Write a note on – SQL facilities.

**AU : May-14, Marks 8**

### 4.11 Two Marks Questions with Answers

**Q.1 What is a transaction ?**

**AU : May-04, Dec.05**

**Ans. :** A transaction can be defined as a **group of tasks** that form a single logical unit.

**Q.2 What does time to commit mean ?**

**AU : May-04**

**Ans. :**

- The COMMIT command is used to save permanently any transaction to database.

- When we perform, Read or Write operations to the database then those changes can be undone by rollback operations. To make these changes permanent, we should make use of **commit**

**Q.3 What are the various properties of transaction that the database system maintains to ensure integrity of data.** **AU : Dec.-04**

**OR**

**Q.4 What are ACID properties ?** **AU : May-05,06,08,13,15,Dec.-07,14,17**

**Ans. :** In a database, each transaction should maintain ACID property to meet the consistency and integrity of the database. These are

(1) Atomicity (2) Consistency (3) Isolation (4) Durability

**Q.5 Give the meaning of the expression ACID transaction.** **AU : Dec.-08**

**Ans. :** The expression ACID transaction represents the transaction that follows the ACID Properties.

**Q.6 State the atomicity property of a transaction.** **AU : May-09,13**

**Ans. :** This property states that each transaction must be considered as a **single unit** and **must be completed fully or not completed at all.**

No transaction in the database is left half completed.

**Q.7 What is meant by concurrency control ?** **AU : Dec.-15**

**Ans. :** A mechanism which ensures that simultaneous execution of more than one transactions does not lead to any database inconsistencies is called **concurrency control mechanism.**

**Q.8 State the need for concurrency control.** **AU : Dec.-17**

**OR**

**Q.9 Why is it necessary to have control of concurrent execution of transactions ? How is it made possible ?** **AU : Dec.-02**

**Ans. :** Following are the purposes of concurrency control –

- To ensure isolation :
- To resolve read-write or write-write conflicts :
- To preserve consistency of database :

**Q.10 List commonly used concurrency control techniques.** **AU : Dec.-11**

**Ans. :** The commonly used concurrency control techniques are –

- i) Lock ii) Timestamp
- iii) Snapshot Isolation

**Q.11 What is meant by serializability? How it is tested?**

**AU : May-14,18, Dec.-14,16**

**Ans. :** Serializability is a concept that helps to identify which non serial schedule and find the transaction equivalent to serial schedule.

It is tested using precedence graph technique.

**Q.12 What is serializable schedule ?**

**AU : May-17**

**Ans. :** The schedule in which the transactions execute one after the other is called serial schedule. It is consistent in nature. For example : Consider following two transactions  $T_1$  and  $T_2$

$T_1$	$T_2$
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

All the operations of transaction  $T_1$  on data items A and then B executes and then in transaction  $T_2$  all the operations on data items A and B execute. The R stands for Read operation and W stands for write operation.

**Q.13 When are two schedules conflict equivalent ?**

**AU : Dec.-08**

**Ans. :** Two schedules are conflict equivalent if :

- They contain the same set of the transaction.
- every pair of conflicting actions is ordered the same way.

For example –

**Non Serial Schedule**

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	

**Serial Schedule**

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)



Write(B)			Write(A)
	Read(B)		Read(B)
	Write(B)		Write(B)

Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

Hence both of the above the schedules are conflict equivalent.

**Q.14 Define two phase locking.**

**AU : May-13**

**Ans. :** The two phase locking is a protocol in which there are two phases :

- i) **Growing Phase (Locking Phase) :** It is a phase in which the transaction may obtain locks but does not release any lock.
- ii) **Shrinking Phase (Unlocking Phase) :** It is a phase in which the transaction may release the locks but does not obtain any new lock.

**Q.15 What is the difference between shared lock and exclusive lock?**

**AU : May-18**

**Ans:**

Shared Lock	Exclusive Lock
Shared lock is used for when the transaction wants to perform read operation.	Exclusive lock is used when the transaction wants to perform both read and write operation.
Multiple shared lock can be set on a transactions simultaneously.	Only one exclusive lock can be placed on a data item at a time.
Using shared lock data item can be viewed.	Using exclusive lock data can be inserted or deleted.

**Q.16 What type of lock is needed for insert and delete operations.**

**AU : May-17**

**Ans. :** The exclusive lock is needed to insert and delete operations.

**Q.17 What benefit does strict two-phase locking provide ? What disadvantages result ?**

**AU : May-06, 07, Dec.-07**

**Ans. : Benefits :**

1. This ensure that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits and preventing other transaction from reading that data .
2. This protocol solves dirty read problem.

**Disadvantage:**

1. Concurrency is reduced.

**Q.18 What is rigorous two phase locking protocol ?**

**AU : Dec.-13**

**Ans. :** This is stricter two phase locking protocol. Here all locks are to be held until the transaction commits.

**Q.19 Differentiate strict two phase locking and rigorous two phase locking protocol.**

**AU : May-16**

**Ans. :**

- In Strict two phase locking protocol all the exclusive mode locks be held until the transaction commits.
- The rigorous two phase locking protocol is stricter than strict two phase locking protocol. Here all locks are to be held until the transaction commits.

**AU : May-08,09,14**

# UNIT - V

## OBJECT RELATIONAL AND NO-SQL DATABASES

### Syllabus

Mapping EER to ODB schema – Object identifier – reference types – rowtypes – UDTs – Subtypes and supertypes – user-defined routines – Collection types – Object Query Language; No-SQL: CAP theorem – Document-based: MongoDB data model and CRUD operations; Column-based: Hbase data model and CRUD operations.

### Mapping an EER Schema to an ODB Schema

It is relatively straightforward to design the type declarations of object classes for an ODBMS from an EER schema that contains *neither* categories *nor*  $n$ -ary relationships with  $n > 2$ . However, the operations of classes are not specified in the EER diagram and must be added to the class declarations after the structural mapping is completed. The outline of the mapping from EER to ODL is as follows:

**Step 1.** Create an ODL *class* for each EER entity type or subclass. The type of the ODL class should include all the attributes of the EER class.<sup>38</sup> *Multivalued attributes* are typically declared by using the set, bag, or list constructors. If the values of the multivalued attribute for an object should be ordered, the list constructor is chosen; if duplicates are allowed, the bag constructor should be chosen; otherwise, the set constructor is chosen. *Composite attributes* are mapped into a tuple constructor (by using a struct declaration in ODL).

Declare an extent for each class, and specify any key attributes as keys of the extent. (This is possible only if an extent facility and key constraint declarations are available in the ODBMS.)

**Step 2.** Add relationship properties or reference attributes for each *binary relationship* into the ODL classes that participate in the relationship. These may be created in one or both directions. If a binary relationship is represented by references in *both* directions, declare the references to be relationship properties that are inverses of one another, if such a facility exists. If a binary relationship is represented by a reference in only *one* direction, declare the reference to be an attribute in the referencing class whose type is the referenced class name. Depending on the cardinality ratio of the binary relationship, the relationship properties or reference attributes may be single-valued or collection types. They will be single-valued for binary relationships in the 1:1 or N:1 directions; they are collection types (set-valued or list-valued) for relationships in the 1:N or M:N direction. An alternative way to map binary M:N relationships is discussed in step 7.

If relationship attributes exist, a tuple constructor (struct) can be used to create a structure of the form <reference, relationship attributes>, which may be included instead of the reference attribute. However, this does not allow the use of the inverse constraint. Additionally, if this choice is represented in *both directions*, the attribute values will be represented twice, creating redundancy.

**Step 3.** Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the original requirements. A constructor method should include program code that checks any constraints that must hold when a new object is created. A destructor method should check any constraints that may be violated when an object is deleted. Other methods should include any further constraint checks that are relevant.

**Step 4.** An ODL class that corresponds to a subclass in the EER schema inherits (via *extends*) the type and methods of its superclass in the ODL schema. Its *specific* (noninherited) attributes, relationship references, and operations are specified, as discussed in steps 1, 2, and 3.

**Step 5.** Weak entity types can be mapped in the same way as regular entity types. An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship; these can be mapped as though they were *composite multivalued attributes* of the owner entity type, by using the `set<struct<... >>` or `list<struct<... >>` constructors. The attributes of the weak entity are included in the `struct<... >` construct, which corresponds to a tuple constructor. Attributes are mapped as discussed in steps 1 and 2.

**Step 6.** Categories (union types) in an EER schema are difficult to map to ODL. It is possible to create a mapping similar to the EER-to-relational mapping (see Section 9.2) by declaring a class to represent the category and defining 1:1 relationships between the category and each of its superclasses. Another option is to use a *union type*, if it is available.

**Step 7.** An  $n$ -ary relationship with degree  $n > 2$  can be mapped into a separate class, with appropriate references to each participating class. These references are based on mapping a 1:N relationship from each class that represents a participating entity type to the class that represents the  $n$ -ary relationship. An M:N binary relationship, especially if it contains relationship attributes, may also use this mapping option, if desired.

## **Object Identifier Types**

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. OIDs are not added to user-created tables, unless `WITH OIDS` is specified when the table is created, or the `default_with_oids` configuration variable is enabled. Type `oid` represents an object identifier. There are also several alias types for `oid`: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, and `regtype`. **Table 8-19** shows an overview.

The `oid` type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. So, using a user-created table's OID column as a primary key is discouraged. OIDs are best used only for references to system tables.

The `oid` type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators. (Beware of possible signed-versus-unsigned confusion if you do this.)

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type `oid` would use. The alias types allow simplified lookup of OID values for objects. For example, to examine the `pg_attribute` rows related to a table `mytable`, one could write

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

rather than

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

While that doesn't look all that bad by itself, it's still oversimplified. A far more complicated sub-select would be needed to select the right OID if there are multiple tables named `mytable` in different schemas. The `regclass` input converter handles the table lookup

according to the schema path setting, and so it does the "right thing" automatically. Similarly, casting a table's OID to regclass is handy for symbolic display of a numeric OID.

**Table 8-19. Object Identifier Types**

Name	References	Description	Value Example
Oid	any	numeric object identifier	564182
Regproc	pg_proc	function name	sum
regprocedure	pg_proc	function with argument types	sum(int4)
Regoper	pg_operator	operator name	+
regoperator	pg_operator	operator with argument types	*(integer,integer) or -(NONE,integer)
Regclass	pg_class	relation name	pg_type
Regtype	pg_type	data type name	integer

All of the OID alias types accept schema-qualified names, and will display schema-qualified names on output if the object would not be found in the current search path without being qualified. The regproc and regoper alias types will only accept input names that are unique (not overloaded), so they are of limited use; for most uses regprocedure or regoperator is more appropriate. For regoperator, unary operators are identified by writing NONE for the unused operand.

An additional property of the OID alias types is that if a constant of one of these types appears in a stored expression (such as a column default expression or view), it creates a dependency on the referenced object. For example, if a column has a default expression nextval('my\_seq'::regclass), PostgreSQL understands that the default expression depends on the sequence my\_seq; the system will not let the sequence be dropped without first removing the default expression.

Another identifier type used by the system is xid, or transaction (abbreviated xact) identifier. This is the data type of the system columns xmin and xmax. Transaction identifiers are 32-bit quantities.

A third identifier type used by the system is cid, or command identifier. This is the data type of the system columns cmin and cmax. Command identifiers are also 32-bit quantities.

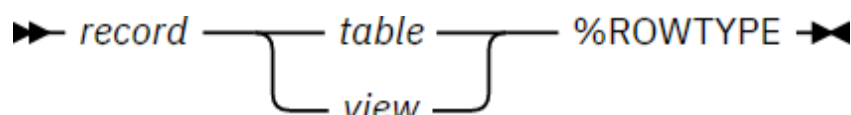
A final identifier type used by the system is tid, or tuple identifier (row identifier). This is the data type of the system column ctid. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.

### **%ROWTYPE attribute in record type declarations (PL/SQL)**

The %ROWTYPE attribute, used to declare PL/SQL variables of type record with fields that correspond to the columns of a table or view, is supported by the Db2® data server. Each field in a PL/SQL record assumes the data type of the corresponding column in the table.

A *record* is a named, ordered collection of fields. A *field* is similar to a variable; it has an identifier and a data type, but it also belongs to a record, and must be referenced using dot notation, with the record name as a qualifier.

### Syntax



Description

**record**

Specifies an identifier for the record.

**table**

Specifies an identifier for the table whose column definitions will be used to define the fields in the record.

**view**

Specifies an identifier for the view whose column definitions will be used to define the fields in the record.

**%ROWTYPE**

Specifies that the record field data types are to be derived from the column data types that are associated with the identified table or view. Record fields do not inherit any other column attributes, such as, for example, the nullability attribute.

Example

The following example shows how to use the %ROWTYPE attribute to create a record (named r\_emp) instead of declaring individual variables for the columns in the EMP table.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno          IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job       : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #    : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee's salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee's salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

## **Reference types**

For every structured type you create, Db2® automatically creates a companion type. The companion type is called a reference type and the structured type to which it refers is called a referenced type. Typed tables can make special use of the reference type. You can also use reference types in SQL statements in the same way that you use other user-defined types. To use a reference type in an SQL statement, use REF(type-name), where type-name represents the referenced type.

Db2 uses the reference type as the type of the object identifier column in typed tables. The object identifier uniquely identifies a row object in the typed table hierarchy. Db2 also uses reference types to store references to rows in typed tables. You can use reference types to refer to each row object in the table.

References are strongly typed. Therefore, you must have a way to use the type in expressions. When you create the root type of a type hierarchy, you can specify the base type for a reference with the REF USING clause of the CREATE TYPE statement. The base type for a reference is called the representation type. If you do not specify the representation type with the REF USING clause, Db2 uses the default data type of VARCHAR(16) FOR BIT DATA. The representation type of the root type is inherited by all its subtypes. The REF USING clause is only valid when you define the root type of a hierarchy. In the examples used throughout this section, the representation type for the BusinessUnit\_t type is INTEGER, while the representation type for Person\_t is VARCHAR(13).

### Reference types

For every structured type you create, Db2 automatically creates a companion type. The companion type is called a reference type and the structured type to which it refers is called a referenced type.

### Relationships between objects in typed tables

You can define relationships between objects in one typed table and objects in another table. You can also define relationships between objects in the same typed table.

### Defining semantic relationships with references

Using the WITH OPTIONS clause of CREATE TABLE, you can define that a relationship exists between a column in one table and the objects in the same or another table.

### Referential integrity versus scoped references

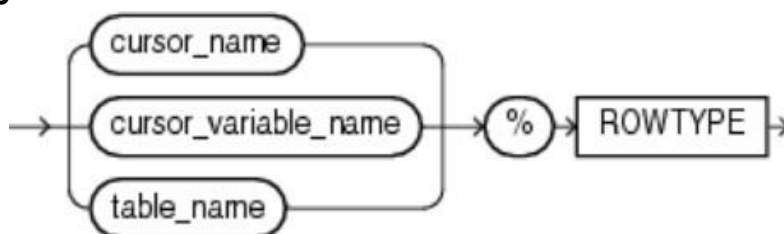
Although scoped references do define relationships among objects in tables, they are different from referential integrity relationships.

## **ROWTYPE Attribute**

The %ROWTYPE attribute provides a record type that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. Fields in a record and corresponding columns in a row have the same names and datatypes.

You can use the %ROWTYPE attribute in variable declarations as a datatype specifier. Variables declared using %ROWTYPE are treated like those declared using a datatype name. For more information, see "[Using the %ROWTYPE Attribute](#)".

### Syntax



```
rowtype_attribute ::=  
{cursor_name | cursor_variable_name | table_name} % ROWTYPE
```

## Keyword and Parameter Description

### **cursor\_name**

An explicit cursor previously declared within the current scope.

### **cursor\_variable\_name**

A PL/SQL strongly typed cursor variable, previously declared within the current scope.

### **table\_name**

A database table or view that must be accessible when the declaration is elaborated.

## Usage Notes

Declaring variables as the type *table\_name*%ROWTYPE is a convenient way to transfer data between database tables and PL/SQL. You create a single variable rather than a separate variable for each column. You do not need to know the name of every column. You refer to the columns using their real names instead of made-up variable names. If columns are later added to or dropped from the table, your code can keep working without changes.

To reference a field in the record, use dot notation (*record\_name.field\_name*). You can read or write one field at a time this way.

There are two ways to assign values to all fields in a record at once:

- First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.
- You can assign a list of column values to a record by using the `SELECT` or `FETCH` statement. The column names must appear in the order in which they were declared. Select-items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases.

## Examples

The following example uses %ROWTYPE to declare two records. The first record stores an entire row selected from a table. The second record stores a row fetched from the `c1` cursor, which queries a subset of the columns from the table. The example retrieves a single row from the table and stores it in the record, then checks the values of some table columns.



```

DECLARE
    emp_rec    employees%ROWTYPE;
    my_empno  employees.employee_id%TYPE := 100;
    CURSOR c1 IS
        SELECT department_id, department_name, location_id FROM departments;
    dept_rec  c1%ROWTYPE;
BEGIN
    SELECT * INTO emp_rec FROM employees WHERE employee_id = my_empno;
    IF (emp_rec.department_id = 20) AND (emp_rec.salary > 2000) THEN
        NULL;
    END IF;
END;
/

```

### **(UDTs ) User Defined Data Type:**

One of the advantages of User Defined Data(UDT) is to Attach multiple data fields to a column. In Cassandra, UDTs play a vital role which allows group related fields (such that field 1, field 2, etc.) can be of data together and are named and type.

In Cassandra one of the advantage of UDTs which helps to add flexibility to your table and data model. we can construct UDT provided by Cassandra: UDT, which stands for User-Defined Type. As we can show in the example that User-defined types (UDTs) can attach multiple data fields in which each named and typed can be mapped to a single column.

To construct User Defined Type (UDT) any valid data type can be used for fields type in Cassandra, including collection ( SET, LIST, MAP) or any other UDTs. Once a UDT in Cassandra is created then it can be used to define a column in the main table.

Syntax to define UDT:

```

CREATE TYPE UDT_name
(
    field_name 1 Data_Type 1,
    field_name 2 Data_Type 2,
    field_name 3 Data_Type 3,
    field_name 4 Data_Type 4,
    field_name 5 Data Type 5,
);

```

#### **Simple steps to create UDTs:**

**Step-1:** Create a KEYSPACE, If not existed. Syntax:

```

create_keyspace_statement ::=
    CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name
    WITH options

```

**Step-2:** To create keyspace used the following CQL query.

```

CREATE KEYSPACE Emp
    WITH replication = {'class': 'SimpleStrategy',
        'replication_factor' : 1};

```

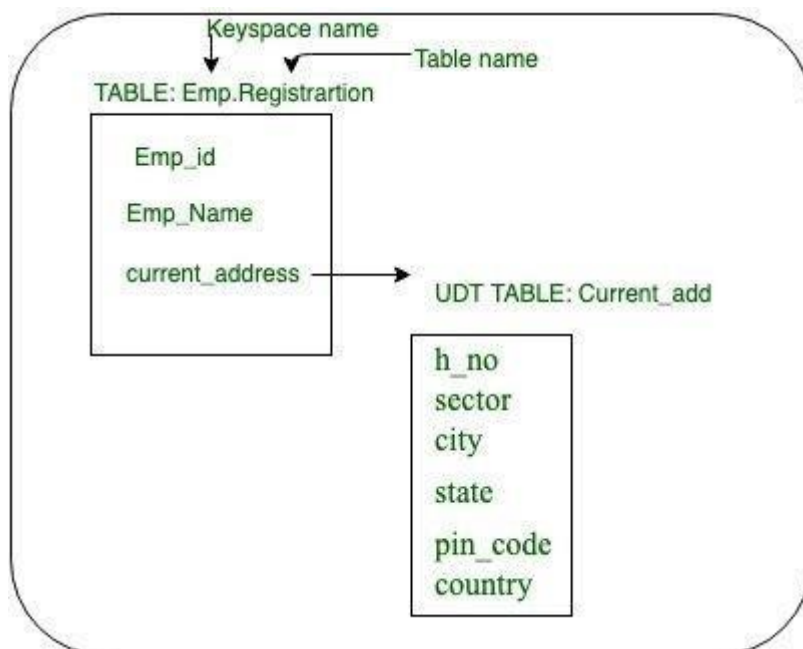
**To check keyspace Schema used the following CQI query.**

```
DESCRIBE KEYSPACE Emp;
```

**Step-3:** To Create Employee User Data Type for Current address used the following CQL query. For example, User Data Type for Current address  
CREATE TYPE Emp.Current\_add

```
(  
Emp_id int,  
h_no text,  
city text,  
state text,  
pin_code int,  
country text  
);
```

Here Current\_add is a Cassandra user-defined data type.



**Figure – User-defined types (UDTs)**

**Step-4:** Create table Registration that is having current\_address UDT as one of the column, for example:

```
CREATE TABLE Registration
```

```
(  
Emp_id int PRIMARY KEY,  
Emp_Name text,  
current_address FROZEN<Current_add>  
);
```

**Step-5:** To insert data using UDT in Cassandra used the following CQL query.

**Format-1: using JSON (JavaScript Object Notation) format.**

In Cassandra we can also insert data in JSON (JavaScript Object Notation) format. For example: CQL query by using JSON format.

```
INSERT INTO Registration JSON'
```

```
{
"Emp_id"          : 2000,
"current_address" : { "h_no" : "A 210", "city" :
                      "delhi", "state" : "DL",
                      "pin_code" : "201307",
                      "country" : "india"},
"Emp_Name"       : "Ashish Rana"}' ;
```

**Format-2: simple insertion**

```
INSERT INTO Registration (Emp_id, Emp_Name, current_address )
      values (1000, 'Ashish', { h_no : 'A 210', city :
                              'delhi', state : 'DL', pin_code
                              :12345, country : 'IN'});
```

```
INSERT INTO Registration(Emp_id, Emp_Name, current_address )
      values (1001, 'kartikay Rana', { h_no : 'B 210 ', city
                                       : 'mumbai', state : 'MH',
                                       pin_code
                                       :12345, country : 'IN'});
```

```
INSERT INTO Registration(Emp_id, Emp_Name, current_address )
      values (1002, 'Dhruv Gupta', { h_no : 'C 210', city
                                       : 'delhi', state : 'DL',
                                       pin_code
                                       :12345, country : 'IN'});
```

**Output:**

	Emp_id	Emp_Name	h_no	city	state	pin code	country
ROW 1	1000	Ashish Rana	A 210	delhi	DL	12345	IN
ROW 2	1001	kartikey Rana	B 210	mumbai	MH	12345	IN
ROW 3	1002	Dhruv Gupta	C 210	delhi	DL	12345	IN

**Figure – UDT Table-insertion**

In this case, we will see how to insert command without inserting the value of one or more fields.

For example, we are not passing the value of the field here. How Cassandra

will handle this?

Well, it will insert this value as a normal value but it will take the field value is null. Every field value, except the primary key that we do not pass at the time of insertion, Cassandra will take it as null.

CQL query without insert one field or more field value of the UDT:

```
INSERT INTO Registration(Emp_id, Emp_Name, current_address )
      values (1003, 'Amit Gupta', { h_no : 'D 210',
city
      : 'Bangalore', state : 'MH',
pin_code :12345}
      );
```

```
INSERT INTO Registration(Emp_id, Emp_Name, current_address )
      values (1004, 'Shivang Rana', { h_no : 'E 210',
city
      : 'Hyderabad', state : 'HYD'}});
```

**Output:**

	Emp_id	Emp_Name	h_no	city	state	pin code	country
ROW 1	1003	Amit Gupta	D 210	Bangalore	MH	12345	null
ROW 2	1004	Shivang Rana	E 210	Hyderabad	HYD	null	null

**Figure – UDT table without insert one or more field value**

## Subtypes and Supertypes

### SuperType and SubType in Data Modeling:

At times, few entities in a data model may share some common properties (attributes) within themselves apart from having one or more distinct attributes. Based on the attributes, these entities are categorized as Supertype and Subtype entities.

**Supertype** is an entity type that has got relationship (parent to child relationship) with one or more subtypes and it contains attributes that are common to its subtypes.

**Subtypes** are subgroups of the supertype entity and have unique attributes, but they will be different from each subtype.

Supertypes and Subtypes are parent and child entities respectively and the primary keys of supertype and subtype are always identical.

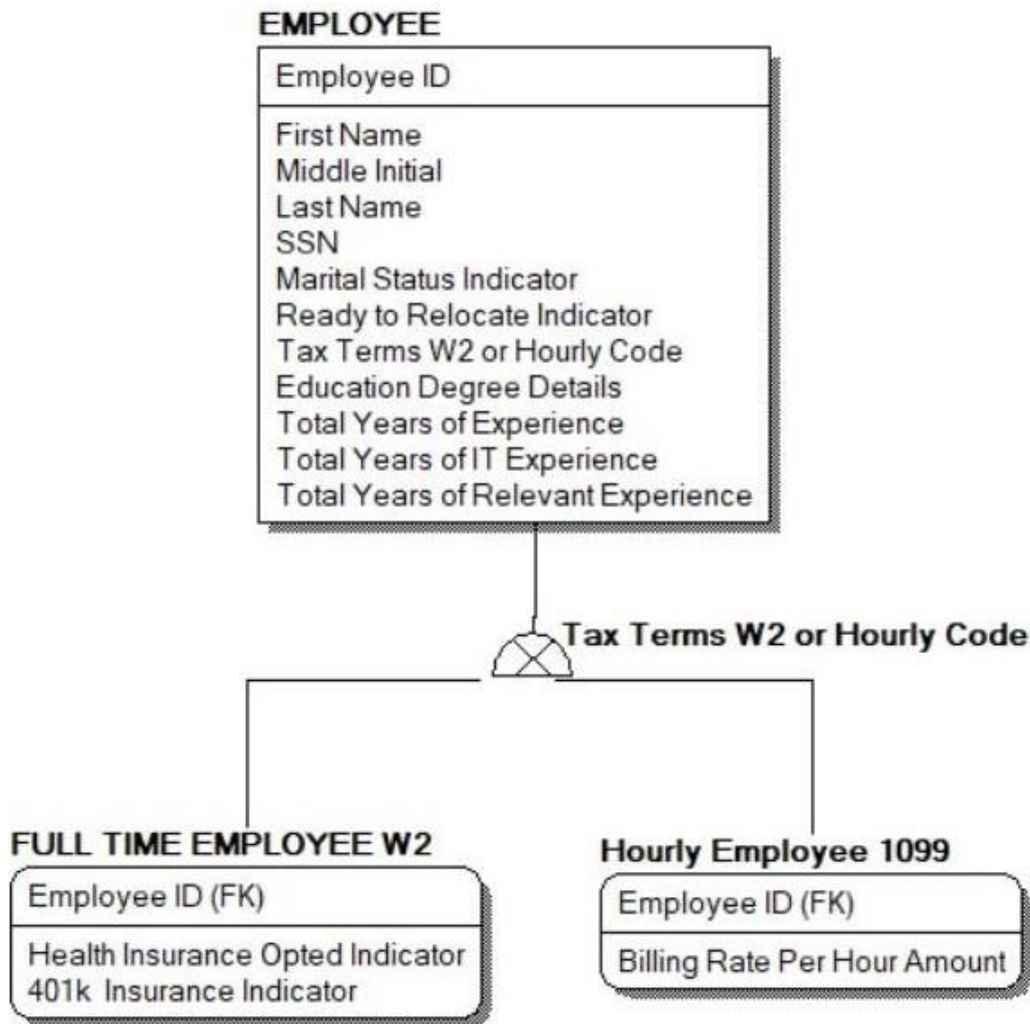
**E.g.** People, Bank Account, Insurance, Asset, Liability, Credit Card.

When designing a data model for PEOPLE, you can have a supertype entity of PEOPLE and its subtype entities can be vendor, customer, and employee. People entity will have attributes like Name, Address, and Telephone number, which are common to its subtypes and you can design entities employee, vendor, and consumer with their own unique attributes. Based on this scenario, employee entity can be further classified under different subtype entities like HR employee, IT employee etc. Here employee will be the superset for the entities HR Employee and IT employee, but again it is a subtype for the PEOPLE entity.

A person can open a savings account or a certificate deposit (fixed deposit) in a bank. These accounts have attributes like account number, account opening date, account expiry date, principal amount, maturity amount, account balance, interest rate, checks issued, pre-cancellation fee etc. While designing a data model, you can create supertype parent entity as "Account" and subtype entities as Savings Account and Certificate Deposit. Account entity will store attributes like account number, interest rate that are common to savings account and certificate deposit entity. Savings account entity will have attributes like account balance and checks issued. While fixed deposit entity will have attributes like account opening, account expiry date, principal amount, maturity amount, pre-cancellation fee etc. When you design a logical data model in this manner, it provides more meaning to the business and the attributes are not cluttered in one table.

### Supertype & Subtype Example:

One good example for explaining this SuperType & SubType is describing the Tax Terms related to Employees. Here Employee is the SuperType or the Parent Entity whereas the two Child Entities "**FULL TIME EMPLOYEE W2**" & "**HOURLY EMPLOYEE 1099**" are the SubTypes.



## User-defined routines

You can create routines to encapsulate logic of your own or use the database provide routines that capture the functionality of most commonly used arithmetic, string, and casting functions. The user-defined routines refer to any procedures, functions, and methods that are created by the user.

You can create your own procedures, functions and methods in any of the supported implementation styles for the routine type. Generally the prefix 'user-defined' is not used when referring to procedures and methods. User-defined functions are also commonly called UDFs.

### User-defined routine creation

User-defined procedures, functions and methods are created in the database by executing the appropriate CREATE statement for the routine type. These routine creation statements include:

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE METHOD

The clauses specific to each of the CREATE statements define characteristics of the routine, such as the routine name, the number and type of routine arguments, and details about the routine logic. The database manager use the information provided by the clauses to identify and run the routine when it is invoked. Upon successful execution of the CREATE statement for a routine, the routine is created in the database. The characteristics of the routine are stored in the database catalog views that users can query. Executing the CREATE statement to create a routine is also referred to as defining a routine or registering a routine.

User-defined routine definitions are stored in the SYSTOOLS system catalog table schema.

### User-defined routine logic implementation

There are three implementation styles that can be used to specify the logic of a routine:

- Sourced: user-defined routines can be *sourced* from the logic of existing built-in routines.
- SQL: user-defined routines can be implemented using only SQL statements.
- External: user-defined routines can be implemented using one of a set of supported programming languages.

When routines are created in a non-SQL programming language, the library or class built from the code is associated with the routine definition by the value specified in the EXTERNAL NAME clause. When the routine is invoked the library or class associated with the routine is run.

User-defined routines can include a variety of SQL statements, but not all SQL statements.

User-defined routines are strongly typed, but type handling and error-handling mechanisms must be developed or enhanced by routine developers.

After a database upgrade, it may be necessary to verify or update routine implementations.

In general, user-defined routines perform well, but not as well as built-in routines.

User-defined routines can invoke built-in routines and other user-defined routines implemented in any of the supported formats. This flexibility allows users to essentially have the freedom to build a complete library of routine modules that can be re-used.

In general, user-defined routines provide a means for extending the SQL language and for modularizing logic that will be re-used by multiple queries or database applications where built-in routines do not exist.

## **PL/SQL - Collections**

we will discuss the Collections in PL/SQL. A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types –

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections –

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

We have already discussed varray in the chapter '**PL/SQL arrays**'. In this chapter, we will discuss the PL/SQL tables.

Both types of PL/SQL tables, i.e., the index-by tables and the nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

## Index-By Table

An **index-by** table (also called an **associative array**) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table\_name**, the keys of which will be of the *subscript\_type* and associated values will be of the *element\_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;
```

```
table_name type_name;
```

### Example

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
```



```

salary_list salary;
name VARCHAR2(20);
BEGIN
-- adding elements to the table
salary_list('Rajnish') := 62000;
salary_list('Minakshi') := 75000;
salary_list('Martin') := 100000;
salary_list('James') := 78000;

-- printing the table
name := salary_list.FIRST;
WHILE name IS NOT null LOOP
  dbms_output.put_line
    ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
  name := salary_list.NEXT(name);
END LOOP;
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

```

Salary of James is 78000
Salary of Martin is 100000
Salary of Minakshi is 75000
Salary of Rajnish is 62000

```

PL/SQL procedure successfully completed.

## Example

Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the **CUSTOMERS** table stored in our database as –

```
Select * from customers;
```

```

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

```

```

DECLARE
CURSOR c_customers is
  select name from customers;

TYPE c_list IS TABLE of customers.Name%type INDEX BY binary_integer;
name_list c_list;
counter integer :=0;

```

```

BEGIN
  FOR n IN c_customers LOOP
    counter := counter +1;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
  END LOOP;
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

```

Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

```

PL/SQL procedure successfully completed

## Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects –

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax –

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
```

```
table_name type_name;
```

This declaration is similar to the declaration of an **index-by** table, but there is no **INDEX BY** clause.

A nested table can be stored in a database column. It can further be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

## Example

The following examples illustrate the use of nested table –

```

DECLARE
  TYPE names_table IS TABLE OF VARCHAR2(10);
  TYPE grades IS TABLE OF INTEGER;
  names names_table;
  marks grades;
  total integer;
BEGIN

```

```

names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
marks:= grades(98, 97, 78, 87, 92);
total := names.count;
dbms_output.put_line('Total '|| total || ' Students');
FOR i IN 1 .. total LOOP
    dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
end loop;
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

```

Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92

```

PL/SQL procedure successfully completed.

## Example

Elements of a **nested table** can also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as –

```

Select * from customers;
+-----+-----+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+-----+-----+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+-----+-----+-----+-----+

```

```

DECLARE
CURSOR c_customers is
    SELECT name FROM customers;
TYPE c_list IS TABLE of customerS.No.ame% type;
name_list c_list := c_list();
counter integer :=0;
BEGIN
FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
END LOOP;
END; /

```

When the above code is executed at the SQL prompt, it produces the following result –

Customer(1): Ramesh  
Customer(2): Khilan  
Customer(3): kaushik  
Customer(4): Chaitali  
Customer(5): Hardik  
Customer(6): Komal

PL/SQL procedure successfully completed.

## **Collection Methods**

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose –

<b>S.No</b>	<b>Method Name &amp; Purpose</b>
1	<b>EXISTS(n)</b> Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.
2	<b>COUNT</b> Returns the number of elements that a collection currently contains.
3	<b>LIMIT</b> Checks the maximum size of a collection.
4	<b>FIRST</b> Returns the first (smallest) index numbers in a collection that uses the integer subscripts.
5	<b>LAST</b> Returns the last (largest) index numbers in a collection that uses the integer subscripts.
6	<b>PRIOR(n)</b> Returns the index number that precedes index n in a collection.
7	<b>NEXT(n)</b> Returns the index number that succeeds index n.
8	<b>EXTEND</b> Appends one null element to a collection.
9	<b>EXTEND(n)</b> Appends n null elements to a collection.
10	<b>EXTEND(n,i)</b> Appends n copies of the i <sup>th</sup> element to a collection.
11	<b>TRIM</b> Removes one element from the end of a collection.

12	<b>TRIM(n)</b> Removes <b>n</b> elements from the end of a collection.
13	<b>DELETE</b> Removes all elements from a collection, setting COUNT to 0.
14	<b>DELETE(n)</b> Removes the <b>n</b> <sup>th</sup> element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If <b>n</b> is null, <b>DELETE(n)</b> does nothing.
15	<b>DELETE(m,n)</b> Removes all elements in the range <b>m..n</b> from an associative array or nested table. If <b>m</b> is larger than <b>n</b> or if <b>m</b> or <b>n</b> is null, <b>DELETE(m,n)</b> does nothing.

## Collection Exceptions

The following table provides the collection exceptions and when they are raised –

Collection Exception	Raised in Situations
COLLECTION_IS_NULL	You try to operate on an atomically null collection.
NO_DATA_FOUND	A subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	A subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	A subscript is outside the allowed range.
VALUE_ERROR	A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a <b>PLS_INTEGER</b> range, and the subscript is outside this range.

## **Object Query Language**

Object Query Language (OQL) is a version of the Structured Query Language (SQL) that has been designed for use in Network Manager. The components create and interact with their databases using OQL.

Use OQL to create new databases or insert data into existing databases (to configure the operation of Network Manager components) by amending the component schema files. You can also issue OQL statements using the OQL Service Provider, for example, to create or modify databases, insert data into databases and retrieve data.

- **Conventions and sample databases**  
To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.
- **Features of OOL**  
The following topics describe the features of Object Query Language (OQL).
- **Database and table creation**  
You can create databases and tables with the `create` command.
- **Inserting data into a table**  
Use the `insert` keyword to insert data into a table.
- **Selecting data from a table**  
You can query the data in a table using the `select` keyword. Use these examples to help you use the `select` keyword.
- **Counting rows in a table**  
You can count the number of rows in a table using the `select` keyword.
- **Conditional tests in OOL**  
Use comparison operators in OQL, for example in a `select where` statement, to perform conditional tests.
- **Use of select to perform subqueries**  
Subqueries are queries that are embedded within queries using double brackets `[[ ]]`. Any valid query can be embedded within the double brackets.
- **Selection of data into another table**  
The `select into` statement retrieves data from one table and inserts it into another. The `select into` command does not delete the existing record.
- **Updates to records in tables**  
Use the `update` keyword to update an existing record in a table.
- **Database and table listings**  
Use the `show` keyword to list the databases, columns, or tables or the current service.
- **Deletion of a record from a database table**  
You can delete a record from a table using the `delete` command.
- **Deletion of a database or table**  
You can delete a database or table using the `drop` command.
- **The eval statement**  
The `eval` statement is used to evaluate the value of a variable or a column within a record, and if necessary, convert it into another data type.

## OQL - Object Query Language

The goal of this file is to help you get started with OQL. The examples presented in this file refer to classes defined in the file "O2 Tutorial".

Please note that the syntax in some of the examples might need minor adjustment before they will work with the current version of O2. If you find any errors, or places which are unclear, or if you have any suggestions or comments, please let us know (email to Michalis - mpetropo@cs.ucsd.edu). Your help is greatly appreciated.

### Introduction

OQL is the way to access data in an O2 database. OQL is a powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

### Using OQL

- Setup your environment
- SELECT, FROM, WHERE
- Dot notation and path expressions
- Subqueries in FROM clause
- Subqueries in WHERE clause
- Set operations and Aggregation
- GROUP BY
- Embedded OQL
- More documentation

### Setup your environment

We've been able to create classes and write some programs. So far, O2 appears to be like an object-oriented programming language like C++ instead of a database system. Probably the main difference is that O2 supports queries. The queries that you'll be creating will look very similar to that of SQL.

In order to perform queries, you'll need to enter query mode. From within the O2 client, do the command:

- **query;**

This will put you in a sub-shell for queries. From here, you can enter your queries followed by Ctrl-D. To exit query mode, just hit Ctrl-D without entering a query.

### SELECT, FROM, WHERE

```
SELECT <list of values>
FROM <list of collections and variable assignments>
WHERE <condition>
```

The SELECT clause extracts those elements of a collection meeting a specific condition. By using the keyword DISTINCT duplicated elements in the resulting collection get eliminated. Collections in FROM can be either extents (persistent names - sets) or expressions that evaluate to a collection (a set). Strings are enclosed in double-quotes in OQL. We can rename a field by if we prefix the path with the desired name and a colon.

### **Example Query 1**

Give the names of people who are older than 26 years old:

```
SELECT SName: p.name
FROM p in People
WHERE p.age > 26
(hit Ctrl-D)
```

### **Dot Notation & Path Expressions**

We use the dot notation and path expressions to access components of complex values.

Let variables *t* and *ta* range over objects in extents (persistent names) of Tutors and TAs (i.e., range over objects in sets Tutors and TAs).

```
ta.salary -> real
t.students -> set of tuples of type tuple(name: string, fee: real) representing
students
t.salary -> real
```

Cascade of dots can be used if all names represent objects and not a collection.

### **Example of Illegal Use of Dot**

*t.students.name*, where *ta* is a TA object.

This is illegal, because *ta.students* is a set of objects, not a single object.

### **Example Query 2**

Find the names of the students of all tutors:

```
SELECT s.name
FROM Tutors t, t.students s
```

Here we notice that the variable *t* that binds to the first collection of FROM is used to help us define the second collection *s*. Because *students* is a



collection, we use it in the FROM list, like t.students above, if we want to access attributes of students.

## Subqueries in FROM Clause

### Example Query 3

Give the names of the Tutors which have a salary greater than \$300 and have a student paying more than \$30:

```
SELECT t.name
FROM ( SELECT t FROM Tutors t WHERE t.salary > 300 ) r, r.students s
WHERE s.fee > 30
```

## Subqueries in WHERE Clause

### Example Query 4

Give the names of people who aren't TAs:

```
SELECT p.name
FROM p in People
WHERE not ( p.name in SELECT t.name FROM t in TAs )
```

## Set Operations and Aggregation

The standard O2C operators for sets are + (union), \* (intersection), and - (difference). In OQL, the operators are written as UNION, INTERSECT and EXCEPT, respectively.

### Example Query 5

Give the names of TAs with the highest salary:

```
SELECT t.name
FROM t in TAs
WHERE t.salary = max ( select ta.salary from ta in TAs )
```

## GROUP BY

The GROUP BY operator creates a set of tuples with two fields. The first has the type of the specified GROUP BY attribute. The second field is the set of tuples that match that attribute. By default, the second field is called PARTITION.

### Example Query 6

Give the names of the students and the average fee they pay their Tutors:

```
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)
FROM t in Tutors, t.students s
GROUP BY sname: s.name
```

## 1. Initial collection

We begin from collection Tutors, but technically it is a bag of tuples of the form:

```
tuple(t: Tutor, s: tuple(name: string, fee: real) )
```

where *t1* is a Tutor object and *s* denotes a student tuple. In general, there are fields for all of the variable bindings in the FROM clause.

## 2. Intermediate collection

The GROUP BY attribute *s.name* maps the tuples of the initial collection to the value of the name of the student. The intermediate collection is a set of tuples of type:

```
tuple( sname: string, partition: set( tuple(t: Tutor, s: tuple( name: string, fee: real ) ) ) )
```

For example:

```
tuple( sname = "Mike", partition = set( tuple(t1, tuple( "Mike", 10 ) ), tuple(t2, tuple( "Mike", 20 ) ) ) )
```

where *t1, t2, ...* are all the tutors of student "Mike".

## 3. Output collection

Consists of student-average fee pairs, one for each tuple in the intermediate collection. The type of tuples in the output is:

```
tuple(sname: string, avgFee: real)
```

Note that in the subquery of the SELECT clause:

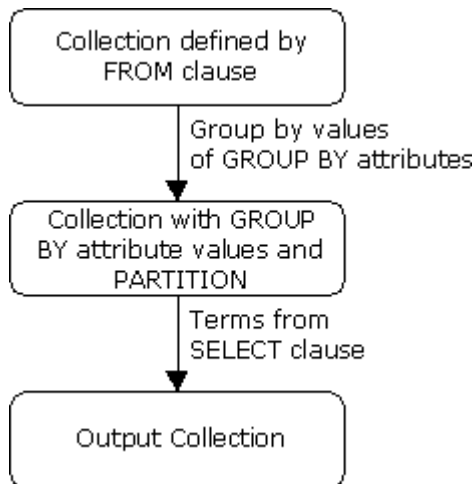
```
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)
```

We let *p* range over all tuples in partition. Each of these tuples contains a Tutor object and a student tuple. Thus, *p.s.fee* extracts the fee from one of the student tuples.

A typical output tuple looks like this:

```
tuple(sname = "Mike", avgFee = 15)
```

The whole procedure of GROUP BY operator's evaluation is summarized in the following figure:



## Embedded OQL

Instead of using query mode, you can incorporate these queries in your O2 programs using the "o2query" command:

```

run body {
  o2 real total_salaries;
  o2query( total_salaries, "sum ( SELECT ta->get_salary \
  FROM ta in TAs )" );
  printf("TAs combined salary: %.2f\n", total_salaries);
};
  
```

The first argument for o2query is the variable in which you want to store the query results. The second argument is a string that contains the query to be performed. If your query string takes up several lines, be sure to backslash (\) the carriage returns.

## **CAP Theorem and NoSQL Databases**

### **What is the CAP theorem?**

The CAP theorem is used to make system designers aware of the trade-offs while designing networked shared-data systems. CAP theorem has influenced the design of many distributed data systems. It is very important to understand the CAP theorem as it makes the basics of choosing any NoSQL database based on the requirements.

CAP theorem states that in networked shared-data systems or distributed systems, we can only achieve at most two out of three guarantees for a database: Consistency, Availability and Partition Tolerance.

A distributed system is a network that stores data on more than one node (physical or virtual machines) at the same time.

Let's first understand C, A, and P in simple words:

**Consistency:** means that all clients see the same data at the same time, no matter which node they connect to in a distributed system. To achieve consistency, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed successful.

**Availability:** means that every non-failing node returns a response for all read and write requests in a reasonable amount of time, even if one or more nodes are down. Another way to state this — all working nodes in the distributed system return a valid response for any request, without failing or exception.

**Partition Tolerance:** means that the system continues to operate despite arbitrary message loss or failure of part of the system. In other words, even if there is a network outage in the data center and some of the computers are unreachable, still the system continues to perform. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

The CAP theorem categorizes systems into three categories:

**CP (Consistent and Partition Tolerant) database:** A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node (i.e., make it unavailable) until the partition is resolved.

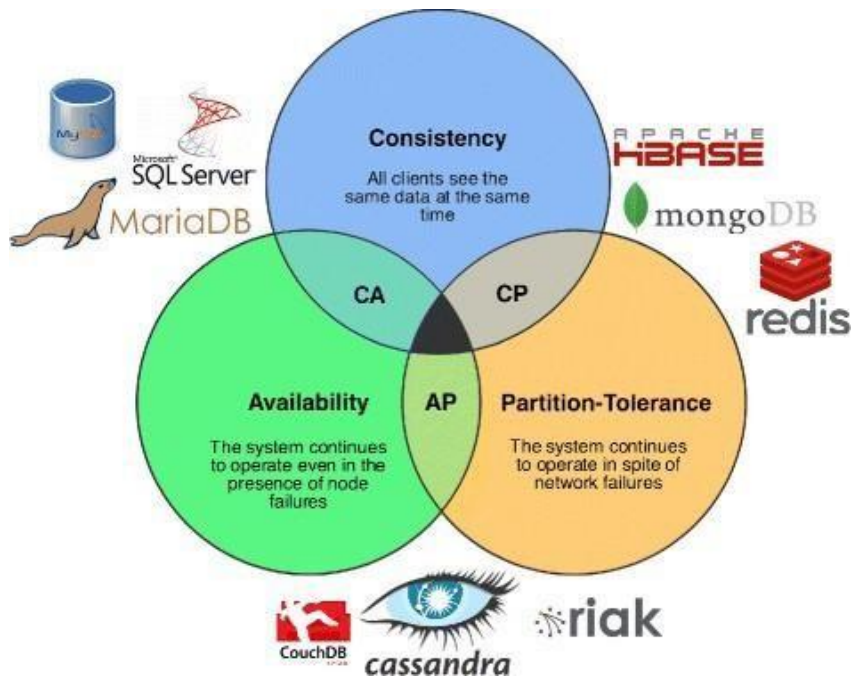
*Partition refers to a communication break between nodes within a distributed system. Meaning, if a node cannot receive any messages from another node in the system, there is a partition between the two nodes. Partition could have been because of network failure, server crash, or any other reason.*

**AP (Available and Partition Tolerant) database:** An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. When the partition is resolved, the AP databases typically resync the nodes to repair all inconsistencies in the system.

**CA (Consistent and Available) database:** A CA delivers consistency and availability in the absence of any network partition. Often a single node's DB servers are categorized as CA systems. Single node DB servers do not need to deal with partition tolerance and are thus considered CA systems.

In any networked shared-data systems or distributed systems partition tolerance is a must. Network partitions and dropped messages are a fact of life and must be handled appropriately. Consequently, system designers must choose between consistency and availability.

The following diagram shows the classification of different databases based on the CAP theorem.



System designers must take into consideration the CAP theorem while designing or choosing distributed storages as one needs to be sacrificed from **C** and **A** for others.

<https://cloudxlab.com/assessment/displayslide/345/nosql-cap-theorem#:~:text=CAP%20theorem%20or%20Eric%20Brewers,data%20at%20the%20same%20time>

## **Data Modeling in MongoDB**

In MongoDB, data has a flexible schema. It is totally different from SQL database where you had to determine and declare a table's schema before inserting data. MongoDB collections do not enforce document structure.

The main challenge in data modeling is balancing the need of the application, the performance characteristics of the database engine, and the data retrieval patterns.

Data in MongoDB has a flexible schema. Documents in the same collection. They do not need to have the same set of fields or structure. Common fields in a collection's documents may hold different types of data.

## **Data Model Design**

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

### **Embedded Data Model**

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

```
{
  _id: ,
  Emp_ID: "10025AE336"
  Personal_details:{
    First_Name: "Radhika",
    Last_Name: "Sharma",
    Date_Of_Birth: "1995-09-26"
  },
  Contact: {
    e-mail: "radhika_sharma.123@gmail.com",
    phone: "9848022338"
  },
  Address: {
    city: "Hyderabad",
    Area: "Madapur",
    State: "Telangana"
  }
}
```

## Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

### Employee:

```
{
  _id: <ObjectId101>,
  Emp_ID: "10025AE336"
}
```

### Personal\_details:

```
{
  _id: <ObjectId102>,
  empDocID: " ObjectId101",
  First_Name: "Radhika",
  Last_Name: "Sharma",
  Date_Of_Birth: "1995-09-26"
}
```

### Contact:

```
{
  _id: <ObjectId103>,
  empDocID: " ObjectId101",
  e-mail: "radhika_sharma.123@gmail.com",
  phone: "9848022338"
}
```

## Address:

```
{
  _id: <ObjectId104>,
  empDocID: " ObjectId101",
  city: "Hyderabad",
  Area: "Madapur",
  State: "Telangana"
}
```

## Considerations while designing Schema in MongoDB

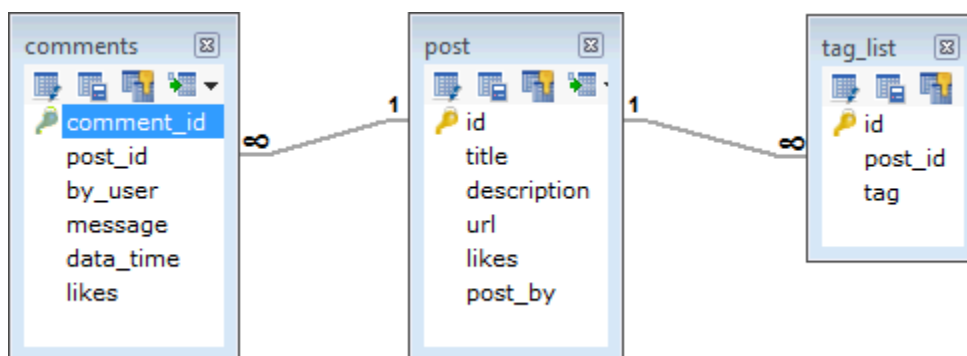
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

## Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure –

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
}
```

```

description: POST_DESCRIPTION,
by: POST_BY,
url: URL_OF_POST,
tags: [TAG1, TAG2, TAG3],
likes: TOTAL_LIKES,
comments: [
  {
    user:'COMMENT_BY',
    message: TEXT,
    dateCreated: DATE_TIME,
    like: LIKES
  },
  {
    user:'COMMENT_BY',
    message: TEXT,
    dateCreated: DATE_TIME,
    like: LIKES
  }
]
}

```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

## **What is CRUD in MongoDB?**

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

### **When it comes to the individual CRUD operations:**

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database.

### **How to Perform CRUD Operations**

Now that we've defined MongoDB CRUD operations, we can take a look at how to carry out the individual operations and manipulate documents in a MongoDB database. Let's go into the processes of creating, reading, updating, and deleting documents, looking at each operation in turn.

### **Create Operations**

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- `db.collection.insertOne()`
- `db.collection.insertMany()`



### *insertOne()*

As the namesake, `insertOne()` allows you to insert one document into the collection. For this example, we're going to work with a collection called `RecordsDB`. We can insert a single entry into our collection by calling the `insertOne()` method on `RecordsDB`. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.RecordsDB.insertOne({
  name: "Marsh",
  age: "6 years",
  species: "Dog",
  ownerAddress: "380 W. Fir Ave",
  chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

### *insertMany()*

It's possible to insert multiple items at one time by calling the `insertMany()` method on the desired collection. In this case, we pass multiple items into our chosen collection (`RecordsDB`) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([
  {
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
  },
  {
    name: "Kitana",
    age: "4 years",
    species: "Cat",
    ownerAddress: "521 E. Cortland",
    chipped: true
  }
])
```

```
db.RecordsDB.insertMany([
  { name: "Marsh", age: "6 years", species: "Dog",
  ownerAddress: "380 W. Fir Ave", chipped: true },
  { name: "Kitana", age: "4 years",
  species: "Cat", ownerAddress: "521 E. Cortland", chipped: true }
])
```

```
"acknowledged" : true,
"insertedIds" : [
  ObjectId("5fd98ea9ce6e8850d88270b4"),
  ObjectId("5fd98ea9ce6e8850d88270b5")
]
```

## Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- `db.collection.find()`
- `db.collection.findOne()`

### *find()*

In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

Here we can see that every record has an assigned “ObjectId” mapped to the “\_id” key. If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

### *findOne()*

In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null.

The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

And, we run the following line of code:

```
db.RecordsDB.find({"age":"8 years"})
```

We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

## Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- `db.collection.updateOne()`
- `db.collection.updateMany()`
- `db.collection.replaceOne()`

*updateOne()*

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "\$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set: {ownerAddress: "451 W. Coffee St. A204"}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

*updateMany()*

*updateMany()* allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

*replaceOne()*

The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

## Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- `db.collection.deleteOne()`
- `db.collection.deleteMany()`

*deleteOne()*

*deleteOne()* is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

*deleteMany()*

*deleteMany()* is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" :
```

## HBase Data Model

The HBase Data Model is designed to handle semi-structured data that may differ in field size, which is a form of data and columns. The data model's layout partitions the data into simpler components and spread them across the cluster. HBase's Data Model consists of various logical components, such as a **table**, **line**, **column**, **family**, **column**, **column**, **cell**, and **edition**.

Row Key	Customer		Sales	
Customer id	Name	City	Product	Amount
101	Ram	Delhi	Chairs	4000.00
102	Shyam	Lucknow	Lamps	2000.00
103	Gita	M.P	Desk	5000.00
104	Sita	U.K	Bed	2600.00

Column Families

### **Table:**

An HBase table is made up of several columns. The tables in HBase defines upfront during the time of the schema specification.

### **Row:**

An HBase row consists of a row key and one or more associated value columns. Row keys are the bytes that are not interpreted. Rows are ordered lexicographically, with the first row appearing in a table in the lowest order. The layout of the row key is very critical for this purpose.

### **Column:**

A column in HBase consists of a family of columns and a qualifier of columns, which is identified by a character: (colon).

### **Column Family:**

Apache HBase columns are separated into the families of columns. The column families physically position a group of columns and their values to increase its performance. Every row in a table has a similar family of columns, but there may not be anything in a given family of columns.

The same prefix is granted to all column members of a column family. For example, Column **courses: history** and **courses: math**, are both members of the column family of courses. The character of the colon (:) distinguishes the family of columns from the qualifier of the family of columns. The prefix of the column family must be made up of printable characters.

During schema definition time, column families must be declared upfront while columns are not specified during schema time. They can be conjured on the fly when the table is up and running. Physically, all members of the column family are stored on the file system together.

### **Column Qualifier**

The column qualifier is added to a column family. A column standard could be **content** (html and pdf), which provides the content of a column unit. Although column families are set up at table formation, column qualifiers are mutable and can vary significantly from row to row.

### **Cell:**

A Cell store data and is quite a unique combination of row key, Column Family, and the Column. The data stored in a cell call its value and data types, which is every time treated as a byte[].

### **Timestamp:**

In addition to each value, the timestamp is written and is the identifier for a given version of a number. The timestamp reflects the time when the data is written on the Region Server. But when we put data into the cell, we can assign a different timestamp value.

## **HBase CRUD Operations**

### ***General Commands***

HBase provides shell commands to directly interact with the Database and below are a few most used shell commands.

**status:** This command will display the cluster information and health of the cluster.

- 1 hbase(main):>status
- 2 hbase(main):>status "detailed"

**version:** This will provide information about the version of HBase.

- 1 hbase(main):> version

**whoami :** This will list the current user.

- 1 hbase(main):> whoami

**table\_help :** This will give the reference shell command for HBase.

- 1 hbase(main):009:> table\_help

### **Create**

Let's create an HBase table and insert data into the table. Now that we know, while creating a table user needs to create required Column Families.

Here we have created two-column families for table 'employee'. First Column Family is 'Personal Info' and Second Column Family is 'Professional Info'.

- 1 create 'employee', 'Personal info', 'Professional Info'
- 2 0 row(s) in 1.4750 seconds
- 3
- 4 => Hbase::Table - employee

Upon successful creation of the table, the shell will return 0 rows.

### **Create a table with Namespace:**

A namespace is nothing but a logical grouping of tables. 'company\_empinfo' is the namespace id in the below command.

- 1 create 'company\_empinfo:employee', 'Personal info', 'Professional Info'

### **Create a table with version:**

By default, versioning is not enabled in HBase. So users need to specify while creating.

Given below is the syntax for creating an HBase table with versioning enabled.

- 1 create 'tableName',{NAME=>"CF1",VERSIONS=>5},{NAME="CF2",VERSIONS=>5}
- 2 create 'bankdetails',{NAME=>"address",VERSIONS=>5}

### **Put:**

Put command is used to insert records into HBase.

- 1 put 'employee', 1, 'Personal info:empId', 10
- 2 put 'employee', 1, 'Personal info:Name', 'Alex'
- 3 put 'employee', 1, 'Professional Info:Dept', 'IT'

Here in the above example all the rows having Row Key as 1 is considered to be one row in HBase.To add multiple rows

- 1 put 'employee', 2, 'Personal info:empId', 20

```
2 put 'employee', 2, 'Personal info:Name', 'Bob'
3 put 'employee', 2, 'Professional Info:Dept', 'Sales'
```

As discussed earlier, the user can add any number of columns as part of the row.

### **Read**

'get' and 'scan' command is used to read data from HBase. Lets first discuss 'get' operation.

**get:** 'get' operation returns a single row from the HBase table. Given below is the syntax for the 'get' method.

```
1 get 'table Name', 'Row Key'
1 hbase(main):022:get 'employee', 1
```

COLUMN	CELL
Personal info:Name	timestamp=1504600767520, value=Alex
Personal info:empId	timestamp=1504600767491, value=10
Professional Info:Dept	timestamp=1504600767540, value=IT

3 row(s) in 0.0250 seconds

### **To retrieve a specific column of row:**

Follow the command to read a specific column of a row.

```
1 get 'table Name', 'Row Key', {COLUMN => 'column family:column'}
2 get 'table Name', 'Row Key' {COLUMN => ['c1', 'c2', 'c3']}
1 get 'employee', 1, {COLUMN => 'Personal info:empId'}
```

COLUMN	CELL
Personal info:Name	timestamp=1504600767520, value=Alex
Personal info:empId	timestamp=1504600767491, value=10
Professional Info:Dept	timestamp=1504600767540, value=IT

3 row(s) in 0.0250 seconds

Note: Notice that there is a timestamp attached to each cell. These timestamps will update for the cell whenever the cell value is updated. All the old values will be there but timestamp having the latest value will be displayed as output.

Get all version of a column

Below given command is used to find different versions. Here 'VERSIONS => 3' defines number of version to be retrieved.

```
1 get 'Table Name', 'Row Key', {COLUMN => 'Column Family', VERSIONS => 3}
```

### **scan:**

'scan' command is used to retrieve multiple rows.

### **Select all:**

The below command is an example of a basic search on the entire table.

```
1 scan 'Table Name'
1 hbase(main):074:> scan 'employee'
```

RO	COLUMN+CELL
1	column=Personal info:Name, timestamp=1504600767520, value=Alex
1	column=Personal info:empId, timestamp=1504606480934, value=15
1	column=Professional Info:Dept, timestamp=1504600767540, value=IT
2	column=Personal info:Name, timestamp=1504600767588, value=Bob
2	column=Personal info:empId, timestamp=1504600767568, value=20
2	column=Professional Info:Dept, timestamp=1504600768266, value=Sales

2 row(s) in 0.0500 seconds

*Note: All the Rows are arranged by Row Keys along with columns in each row.*

### Column Selection:

The below command is used to Scan any particular column.

```
1 hbase(main):001:>scan 'employee' ,{COLUMNS => 'Personal info:Name' }
```

```
RO                                COLUMN+CELL
1                column=Personal info:Name, timestamp=1504600767520, value=Alex
2                column=Personal info:Name, timestamp=1504600767588, value=Bob
2 row(s) in 0.3660 seconds
```

### Limit Query:

The below command is used to Scan any particular column.

```
1 hbase(main):002:>scan 'employee' ,{COLUMNS => 'Personal info:Name',LIMIT =>1 }
```

```
RO                                COLUMN+CELL
1                column=Personal info:Name, timestamp=1504600767520, value=Alex
1 row(s) in 0.0270 seconds
```

### Update

To update any record HBase uses 'put' command. To update any column value, users need to put new values and HBase will automatically update the new record with the latest timestamp.

```
1 put 'employee', 1, 'Personal info:empId', 30
```

The old value will not be deleted from the HBase table. Only the updated record with the latest timestamp will be shown as query output.

To check the old value of any row use below command.

```
1 get 'Table Name', 'Row Key', {COLUMN => 'Column Family', VERSIONS => 3 }
```

### Delete

'delete' command is used to delete individual cells of a record.

The below command is the syntax of delete command in the HBase Shell.

```
1 delete 'Table Name' , 'Row Key', 'Column Family:Column'
```

```
1 delete 'employee',1, 'Personal info:Name'
```

### Drop Table:

To drop any table in HBase, first, it is required to disable the table. The query will return an error if the user is trying to delete the table without disabling the table. Disable removes the indexes from memory.

The below command is used to disable and drop the table.

```
1 disable 'employee'
```

Once the table is disabled, the user can drop using below syntax.

```
1 drop 'employee'
```

You can verify the table in using 'exist' command and enable table which is already disabled, just use 'enable' command.