Introduction to DBMS

Designing and Implementing Databases from Scratch for Absolute Beginners

DR. HARIRAM CHAVAN

PROF. SANA SHAIKH

Introduction to DBMS

Designing and Implementing Databases from Scratch for Absolute Beginners

DR. HARIRAM CHAVAN

PROF. SANA SHAIKH

bpb

Introduction to DBMS

Designing and Implementing Databases from Scratch for Absolute Beginners

Dr. Hariram Chavan
Prof. Sana Shaikh

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: May 2022

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55510-266

www.bpbonline.com

Dedicated to

Our Families

About the Authors

- **Dr. Hariram Chavan** is currently working as a Professor in the Department of Information Technology of K. J. Somaiya Institute of Engineering and Information Technology, Mumbai having a rich experience of 25 years in teaching along with six years of experience in academic administration. He has been a Member of the syllabus setting committee at the University of Mumbai and designed the syllabus for various courses at UG and PG levels. He also filed a patent for his research work in Mobile databases using Machine Learning work. He always strives to solve real-world problems through his technical skills. To keep updated himself he has earned certifications from a few MOOCs on the database, data science, machine learning, deep learning, artificial intelligence, and related courses.

- **Prof. Sana Shaikh** is currently working as an Assistant Professor in the Department of Computer Engineering of Don Bosco Institute of Technology, having fourteen years of teaching experience along with three and half years of experience in academic administration. She has been a member of the syllabus setting committee at the University of Mumbai and designed the syllabus for various courses at UG levels. She earned her Master's Degree in Computer Science from the University of Mumbai, India, and has a wide range of interests, covering topics such as Database Management Systems, Web Mining, Geo-informatics, and Big Data Analytics. She has also published many research papers in IEEE, Springer conferences, and in reputed journals. She has received a research grant from Mumbai University for her research project. To keep herself updated, she has earned certifications from a few MOOCs on Python, Big Data Analytics, Cloud Computing, and related courses.

About the Reviewer

With more than 21 years of experience in academia and industry, **Dr. R Rajkumar** is currently working at RNS Institute of Technology, Bengaluru as an Associate Professor. He has vast experience in teaching and mentoring both undergraduate and postgraduate students in subjects like Computer Organization, UNIX Programming, Database Management Systems, Digital Communication, Internet of Things, Java Programming, etc. He has several publications in refereed journals on Deep Learning techniques. His current area of interest is the Optimization of Algorithms for Data and Intelligent Systems.

Acknowledgement

It is with great pleasure the authors acknowledge that this book would not have been possible without the collaboration of numerous people whom we wish to thank and remember.

We are grateful for the support of my friends, colleagues, and my family. We are also appreciative of the contribution extended by these reviewers and their helpful suggestions:

Prof. Smita Dange and Prof. Neeraj Khairwal

Finally, we would like to thank the editorial team of BPB Publications involved with this book for their encouragement and sustenance.

Preface

To the reader,

This book used creative approaches for their readers to ask questions independently, make connections between ideas, think creatively, challenge and participate effectively, and reflect on their learning. It is a comprehensive book covering the basic concepts of Database Management Systems. It is specially written for those who are looking for an effective source to learn the concepts of database design which are presented through real-time examples.

The content of this book is inclined to be an introductory course in database systems as it is an integral part of computer science and engineering. The text book is organized in a manner suitable for one-semester undergraduate and graduate database courses. We have attempted to present the contents in a simple, clear, and lucid style. The readers will be able to:

- Learn and practice data modeling using the entity-relationship and relational models.
- Understand the use of Structured Query Language (SQL) and emphasize the in-depth treatment of SQL queries.
- Apply the normalization process and its importance in the database design.
- Understand the need of database processing and learn techniques for controlling the consequences of concurrent data access.

This book is written in such a manner (using examples of the actual world) so that readers can link to various applications for their better understanding. A quantitative approach is used for exercises of each chapter and it will reinforce readers' ability to apply the concepts while attempting to solve the real problems. The format of each chapter is as follows- Introduction. Concepts. Summary. Forward to the next chapter and Exercise.

Chapter 1 introduces the basic concepts of database systems. The flaws of the file system and the advantages of the database system are listed. The data abstraction, data independence, and database system architecture with their functions are presented. The interaction of the different types of users and their roles are explained.

Chapter 2 introduces the concepts of data modeling through the Entity-Relationship (ER) model which defines the conceptual view of a database. It works around real-world entities and associations among them. The problem statement for the engineering college database is used throughout the text to present the various concepts of ER modeling. The mapping relationship, constraints on relationships, and participation in relationships are explained with examples. It also introduces the concept of generalization/specialization and concludes with the steps of Entity Relationship Diagram (ERD) generation.

Chapter 3 encompasses the relational model and relational algebra. It will begin with the fundamental concepts of Relational Models and define the different keys. The concepts of integrity constraints and their importance in the database models are illustrated with examples. The second section of this chapter elaborates on the operators, operations, and relational algebra queries. It concludes with mapping the ER and Extended Entity Relationship (EER) Model to relational schema and justifies how data redundancy is handled in the relational database models.

Chapter 4 will focus on SQL syntax and using the SQL commands to obtain the desired results. The Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), Transaction Control Language (TCL), and string operations are explained using MySQL and the engineering college database. The aggregate functions to generate summarized results, implementation of a cursor, triggers, and embedded SQL using Java Database Connectivity (JDBC) are elaborated with simple examples.

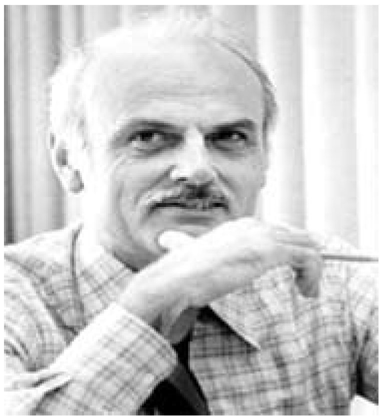
Chapter 5 focuses on functional dependency, functional decomposition, and the synthetic approach of relational database design. The basic normal forms and the process of normalization are demystified. It illustrates the good table structure, database anomalies, and organization of the data into logical groups that store the information without unnecessary redundancy.

Chapter 6 defines a transaction and describes how you can manage your work using transactions and their ACID (**A**tomicity, **C**onsistency, **I**ntegrity, **D**urability) properties. The problems associated with the concurrent execution of transactions and the various schemes used to resolve them with deadlock are presented. Various methods of recovery from the loss of data are explained in case of failure of the database system.

Table of Contents

CHAPTER 1

Database System Concepts and Architecture



Edgar Codd
(23 August 1923 - April 18, 2003)

We all must be thankful for the efforts of Edgar (Ted) Frank Codd, the Royal Air Force pilot. He was the pilot who created the theories of data management based on the mathematical set theory. His idea of data storage using a simple table structure bought a technological revolution. "*A Relational Model of Data for Large Shared Data Banks*", the landmark publication of Codd talks about representation, abstraction, relation, and normal form of data.

We are living in the digital era where the word '*data*' has platinum importance. It became an integral part of our daily lives. The organization which acquires timely accurate data may rule the world. Data processing has changed from what happened, why it happened, and what will happen, to the most recent stage, can we make it happen. This is the power of data. So, in this chapter, we will focus on data and the related terms.

Introduction

Data, database, and database technology have played a critical role in every field. It has a major impact on the growth of every business. By data, we mean "*raw facts*". Here "*raw*" means unprocessed. The dictionary meaning of "*fact*" is "*a thing that is known or proved to be true*". In other words, data carries some inherent meaning associated with it and can be revealed by processing.

Data: Raw facts

The following are some raw facts on data:

- Data means known truth about anything which exists in the real world that can be stored for future use and also has some implicit meaning.
- Data is raw, unorganized facts that need to be processed. Data can be something simple and seemingly random and useless until it is organized.
- For example, the Test scores of a student, Enrolment Number, Email ID, and so on.

Structure

In this chapter, we will cover the following topics:

- Difference between data and information
- Understanding the basic concepts of database systems
- Issues with the file-based system
- Major components of the database management system and their functions
- Understanding the role of the database users and database administrator

Objectives

After studying this chapter, you will be able to understand data, information, database, and DBMS, differentiate between data and information, and outline why the database design is important. You will also be able to list the flaws of the file system, recall the main components of the database system with their functions, state the role of the database users, and database administrator.

Information: Processed data

This is the technical definition of the information. But in reality, what is information? To understand the meaning of information, we will consider a few examples. Let us consider that you are speaking with your friend and the dialog is "*It is raining here*". This statement carries information if the season is other than rainy. Since, if the season is rainy, then rainfall is expected and does not carry any information. Instead, if the statement is something like "*It is raining heavily*", then it may carry some information considering the current scenario of rainfall in India. One more example to understand information is the human interest in the uneven happening and the news based on that. For example, hundreds of planes take off safely or land safely every day. But it never became a piece of news. The moment there is a problem during take-off or landing, it becomes a piece of news as it carries information. This means that information is something new that you will get after processing the data or occurrence of an event.

The following are some more facts on information:

- Data can be converted into a meaningful and useful context. Later, it can be communicated to a recipient who uses it to make the decisions.
- When the data is organized, processed, and presented in a given context, it becomes information.
- For example, the subject teacher can process the test scores of the students, and classify them into weak and bright students.

Database

A database is a collection of interrelated data.

Now, answer the following questions in the YES/NO form:

- How many of you have recorded friends' birthdays in a diary?

- How many of you have written colleagues' addresses into an address book?
- How many of you have referred to a dictionary for new words?

If you have answered YES to any of these questions, then you have already used a paper-based database in your daily life, where the following happened:

- Birthdays were organized in the logical order of '*Month*'.
- Addresses were stored in a logical order of '*Name*'.
- English words with their meaning are arranged in alphabetical order.

These were examples of the paper-based databases. However, when we use the term '*database*', we generally think of a computerized database. Actually, without realizing, we are all using databases almost every day in our life.

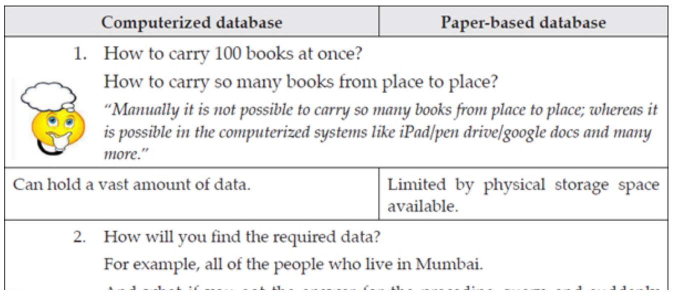
Refer to the following as examples:

- Google Search Engine
- ATM
- Online food ordering
- Online shopping
- Online booking flight

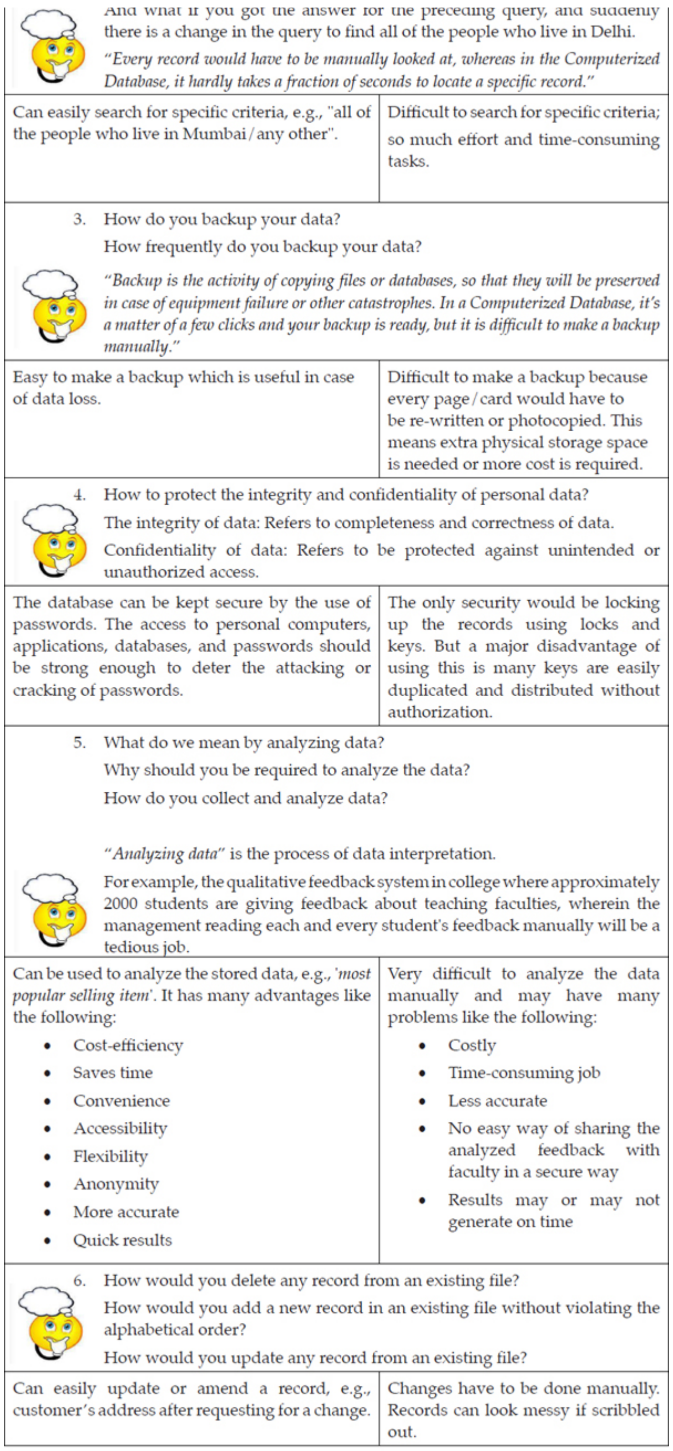
Computerized versus paper-based databases

What is it that makes the computerized databases much more popular than the paper-based ones?

Refer to *Table 1.1* as follows:



Computerized database	Paper-based database
1. How to carry 100 books at once? How to carry so many books from place to place? *"Manually it is not possible to carry so many books from place to place; whereas it is possible in the computerized systems like iPad/pen drive/google docs and many more."*	
Can hold a vast amount of data.	Limited by physical storage space available.
2. How will you find the required data? For example, all of the people who live in Mumbai.	



And what if you got the answer for the preceding query, and suddenly there is a change in the query to find all of the people who live in Delhi. *"Every record would have to be manually looked at, whereas in the Computerized Database, it hardly takes a fraction of seconds to locate a specific record."*	
Can easily search for specific criteria, e.g., "all of the people who live in Mumbai / any other".	Difficult to search for specific criteria; so much effort and time-consuming tasks.
3. How do you backup your data? How frequently do you backup your data? *"Backup is the activity of copying files or databases, so that they will be preserved in case of equipment failure or other catastrophes. In a Computerized Database, it's a matter of a few clicks and your backup is ready, but it is difficult to make a backup manually."*	
Easy to make a backup which is useful in case of data loss.	Difficult to make a backup because every page / card would have to be re-written or photocopied. This means extra physical storage space is needed or more cost is required.
4. How to protect the integrity and confidentiality of personal data? The integrity of data: Refers to completeness and correctness of data. Confidentiality of data: Refers to be protected against unintended or unauthorized access.	
The database can be kept secure by the use of passwords. The access to personal computers, applications, databases, and passwords should be strong enough to deter the attacking or cracking of passwords.	The only security would be locking up the records using locks and keys. But a major disadvantage of using this is many keys are easily duplicated and distributed without authorization.
5. What do we mean by analyzing data? Why should you be required to analyze the data? How do you collect and analyze data? "*Analyzing data*" is the process of data interpretation. For example, the qualitative feedback system in college where approximately 2000 students are giving feedback about teaching faculties, wherein the management reading each and every student's feedback manually will be a tedious job.	
Can be used to analyze the stored data, e.g., '*most popular selling item*'. It has many advantages like the following: • Cost-efficiency • Saves time • Convenience • Accessibility • Flexibility • Anonymity • More accurate • Quick results	Very difficult to analyze the data manually and may have many problems like the following: • Costly • Time-consuming job • Less accurate • No easy way of sharing the analyzed feedback with faculty in a secure way • Results may or may not generate on time
6. How would you delete any record from an existing file? How would you add a new record in an existing file without violating the alphabetical order? How would you update any record from an existing file?	
Can easily update or amend a record, e.g., customer's address after requesting for a change.	Changes have to be done manually. Records can look messy if scribbled out.

Table 1.1: *Computerized vs. paper-based databases*

Database Management System (DBMS)

The database management system is a collection of control routines (software package) that manages the data functionality, mainly for easy storage and easy retrieval. In reality, it does many other important tasks from the creation of a database to the maintenance of a database which in turn includes the following:

- **Data definition**: It involves specifying the data types, data structures, and constraints of the data to be stored.
- **Data storage**: Storing data mostly on secondary storage (or server) which is controlled by DBMS.
- **Data manipulation**: The three main informal operations are add, modify, and delete; technically inserting, querying, updating and generating reports.
- **Concurrent access**: The main advantage of the database management systems is concurrent access or data sharing which allows multiple users and programs to access the data concurrently.
- **Data protection**: The DBMS provides protection against the unauthorized access, system failure, and other catastrophic failures.

Characteristics of databases

The database has many interesting features which makes it more suitable and useful than the file systems. Before the database system, the common or traditional approach to manage the data was the file system. But as we know that the file system has many limitations, we switched to the database systems to avail the benefits of it. To understand the limitations of the file systems, let us consider a simple example.

Let us assume that the engineering college does not have a centralized database system that maintains the enrolled students. In an engineering college, a student enrolls through a centralized admission process. The Director of Technical Education (DTE) gives a list of students who have registered themselves for a particular program in a particular engineering college. Also, take into consideration that this list is in an Excel format. Now, consider that the administration office (Accounts department) maintains this list for fee collection in tally software. Then, this data will be given to the first-year engineering department/humanities department. The department maintains the list in an Excel format as well. In the second year, the student’s data will be maintained by individual departments. Now, let us assume that the IT department uses MS Access to create and maintain the student's data. The computer department uses Oracle for the same purpose. According to this scenario, the same database is available in various formats in different departments.

Now, if any student changes their mobile number, then it has to be reflected in at least three different files. That is in the administration office (Accounts department), humanity department, the actual department, and Training & Placement department, etc. If any one of the files is not updated, then the data becomes inconsistent. To add to the complexity, assume that the Training & Placement department maintains the student data for different companies based on company requirements and they have their own tailor-made software which generates the student report based on the companies' criteria. As we have considered the different file systems, it will increase the inconsistency with each data change which is not reflected at all places/files. If you consider that the phone number, email address, and student address might frequently change, with every field, the data inconsistency increases.

Other than data inconsistency, another major problem that arises in case of a separate file system is the data structure that holds the data and the naming system used by different file systems (i.e., creator of the file). In the preceding example, let us consider the phone number field of the student database. In the administration office (Accounts department), the field name is mobile number, in the Information Technology department, it has been referred to as contact number. The Computer department stores both landline and mobile number under the heads land and hand respectively. Now, if the information about these fields is not maintained, then again it will create confusion. So, to limit the data inconsistency, to avoid misperception, to maintain the uniformity, we must maintain the data at one place and make it available to all those who would like to access it, and for this, we require a database management system.

In the case of database management systems, it maintains a single repository and will be made available to all users. In the preceding case, the administration department (Accounts department), the humanities department, the individual department, and the Training & Placement cell. If there is any change in any field, then it will be made in one place and will be available for all the users, so that there is no possibility of inconsistency and misinterpretation of the data.

The main characteristics of the database approach are as follows:

- Self-describing nature of a database system
- Programs and data insulation
- Multiple views of data
- Data sharing

We will describe each of these characteristics in detail in the following section.

Self-describing nature of a database system

A database system consists of data and meta-data. The meta-data is the data about the data. Meta-data provides complete information about the structure and constraints of the data. It can be used by the user to know the type, structure, and constraints of each data item. This feature makes the database system different from the traditional file system. In a file-based system, the data definition is part of the application program. A database system is a general-purpose software package. Therefore, it can be used to model the different applications. The history shows that the database will work equally well for database applications such as student management systems, hospital management systems, hotel management systems, etc. The most prominent use that we can experience in our day-to-day life is the banking system. It truly reduces the burden of the bank employees to maintain different ledgers. The tailor-made software automatically creates the report desired by the bank manager.

This feature makes the database system completely different from the traditional file system. In a file-based system, the data definition is part of the application program. For example, in the case of C programming, the data required may be declared separately or using struct. The database software provides the drivers to access these specific databases.

Program and data insulation

One of the main advantages of a database system is data independence. The database system insulates the data from the application program. In contrast, in traditional file-processing systems, the data is part of the application programs. This means that the user can change the structure of the program without changing the structure of the data.

In general, the database systems are composed of complex data structures. For making an efficient and user-friendly system, the developers need to hide the complexity of the system from the end-users. The process of hiding the complex details of the system from the user is known as "*Data Abstraction*". This approach simplifies the database design. There are mainly three levels of data abstraction, namely **Physical**, **Logical**, and **View Level**, as shown in *Figure 1.1*:

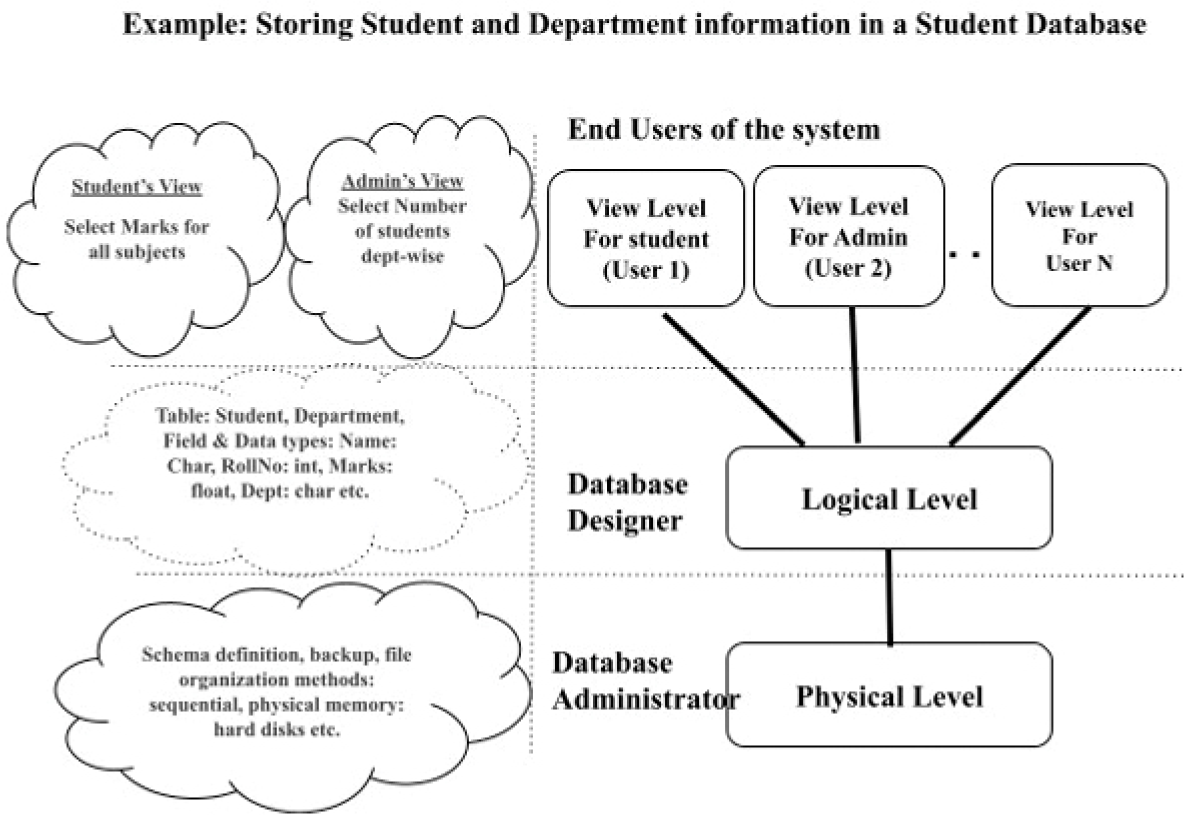

Example: Storing Student and Department information in a Student Database
Student's View
Select Marks for all subjects
Admin's View
Select Number of students dept-wise
End Users of the system
View Level For student (User 1)
View Level For Admin (User 2)
View Level For User N
Table: Student, Department, Field & Data types: Name: Char, RollNo: int, Marks: float, Dept: char etc.
Database Designer
Logical Level
Schema definition, backup, file organization methods: sequential, physical memory: hard disks etc.
Database Administrator
Physical Level

Figure 1.1: *Data abstraction levels*

Physical level

This is the lowest level in data abstraction. The physical level outlines the low-level data structures in detail. The database designer can plan the data structures considering the future requirements of data storage.

This level describes the following factors that the database administrators need to know while designing the database:

- How is the data actually stored in the physical memory (like magnetic tapes, hard disks, etc.)?
- Which file organization method has been used (like hashing, sequential, B+ tree, etc.)?
- What will be the blocks of storage (bytes, kilobytes, megabytes, gigabytes, etc.) in memory?

Logical or Conceptual level

This is the next level of data abstraction. The database designer decides the data to be stored in the database based on the application requirements. The database designer designs a complete database using simple relations based on the normalization (more about normalization in Chapter 5, *Relational Database Design*) concept.

The logical level describes what data are stored in the database and the relations among them. It describes the interrelated data in terms of a bunch of simple relations with relatively simple structure.

The database designer may plan the complex physical level structure while designing the database but the user need not worry about this complexity. Since, the users at a logical level are unaware of the complex physical level structure, it is referred to as physical data independence.

View level

This is the final level of data abstraction. It describes only part of the entire database. The end users access only part of the database based on the defined roles.

The view level hides the complex details (storage and implementation details) as well as the logical schema from the end-users. In the case of complex databases, the complexity is reduced by creating/providing multiple views of the database.

The end users generally interact with the database through user-friendly **Graphical User Interface** (**GUI**). View level simplifies the end-user's data access.

For example, consider the students' information is stored in a **Student** and **Department** tables, as shown in *Figure 1.1*.

Physical Level: **Database Administrator** (**DBA**) decides data structures based on schema definition and optimized responses expected. He also decides the frequency of backup, the suitable file organization method, etc. For example, in the case of the **Student** database, the following happens:

- Understanding how the student's data is actually stored in the physical memory
- Maintaining daily backup
- Creating indexes for faster data access

Logical level: At this level, the database administrator will design the conceptual/logical schema of the database. DBA also decides the number of relations, their interrelations, integrity constraints, etc. For example, in the case of **Student** database, the following are taken care of:

- Student and Department relations
- Student: Name - Varchar, RollNo - int, Marks - float etc.
- Department: Did - Varchar, Dname - Varchar, etc.
- Student's enrollment in the department (Many: One)
- Constraint: Students can enroll in one department only.

View level: DBA is one of the core users at this level. He will decide the roles and responsibilities of the different database users. For example, in the case of the **Student** database, the following happens:

- End users (Students, Accountants, Training & Placement Officer (TPO), etc.) will access the database through interactive GUI. Here, the students can view the marks scored in different subjects but can't change.

- Accountants can view and update the fee status of the students.
- TPO can generate the student's reports
    - Placed students' list to the management
    - Eligible students' list company-wise
    - Admin can create, view, delete, and update the student's details

Multiple views of data

A view is a subset of the database. The view is defined based on the end-user role. The multiple views are created for multiple users. Each view may contain a subset of relations or composition of relations based on a group of users.

The following are the examples:

- The account section need only the part of the student database.
- The department requires a list of students.
- The training and placement cell may be interested in the percentage or pointer of the students.

So, based on the requirements of these end users, different views may be created.

Data sharing

One of the most important advantages of a database system is concurrent/simultaneous access of data. The database systems ensure concurrency control strategies for simultaneous access of data. The main role of concurrency control is when many users are trying to update the same data. The database system should ensure data integrity, i.e., the accessed data is always correct; for example, the railway reservation system.

Several users might use the *Railway Reservation System* at the same time to reserve a seat. In this case, the underlying database system should ensure that each seat can be accessed by only one user at a time.

File system versus Database system

Earlier, the file system was the most popular way to store and organize the data on drives. However, nowadays, when it comes to security and organizing of the data including constraints, many experts choose **Database Management System (DBMS)**.

Refer to *Table 1.2*, as follows:

Characteristics	**Database approach**	**File processing approach**
Type of application	Data definition is specified using DDL. This is a general-purpose program, not specific to any application.	Data definition is typically part of the application programs themselves. So it works with a specific database.
Security	1. Security can be provided by using a password. 2. Security can be achieved by assigning roles to the end-users using grant and revoke.	Data is less secure in the file processing systems as the entire file can be protected through a password but not part of it.
Concurrency control	1. Multi-user access control. 2. Concurrent access anomalies can be removed using different concurrency control algorithms.	Concurrent access may generate problems.
Isolation	1. Data isolation is possible as data will be in a single format. 2. Isolation is achieved through Abstraction levels.	Data isolation is not possible as data will be in different formats.

Integrity or consistency constraints	1. Different integrity constraints can be enforced such as entity integrity constraints, ... etc. 2. Integrity constraints help maintain consistent data. 3. Very less redundancy and which is under control or controlled data redundancy.	1. Enforcing integrity constraints is very difficult. 2. Application programmers have to take care of it.
Data redundancy and inconsistency	1. Special algorithms can be developed and used to ensure data consistency. 2. Data redundancy can be reduced by using specialization or generalization constructs.	1. Because of different types of formats and different programmers, it may lead to data inconsistency. 2. It is difficult to ensure data consistency.
Accessing data	1. Data access is easy and efficient 2. Data access using SQL. 3. Report generation for naive users.	1. Data access is a very time-consuming task. 2. Special application programs are required to access data.

Table 1.2: *Difference between File System and DBMS*

The Three-Schema Architecture

There are three types of database architecture, which are as follows:

- 1 tier
- 2 tier
- 3 tier

1-tier database architecture

It is the simplest database architecture. When a single machine consists of the client, server, and database, it is called 1-tier database architecture, as shown in *Figure 1.2*. As the 1-tier database architecture compromises the security, it is absolute nowadays; for example, a Microsoft Excel spreadsheet (used as a file system).

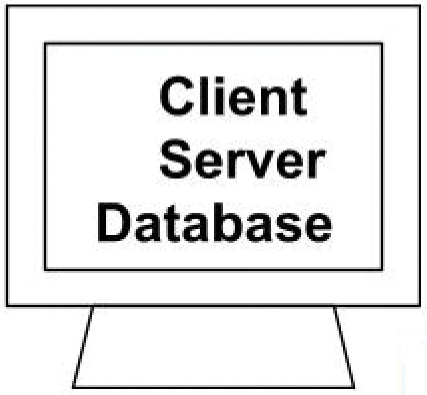


Figure 1.2: *1-tier architecture*

2-tier database architecture

A two-tier architecture is a database architecture where the client is directly interacting with the server, as shown in *Figure 1.3*. The presentation layer runs on a client machine. The data is stored on the database server.

For example, in the late '90s, in small organizations or enterprises, the client machine interacts directly with the database server but it supports only a limited number of clients. Due to this, it suffers from scalability and performance issues.

The client machine interacts with the server through an interface to fetch the data from the server. The application interface is called Open Database Connectivity (ODBC). This API allows the client-side program to call the DBMS.

Today, various ODBC drivers are available for DBMS. The 2-tier architecture provides end-user access through user authorization, as it is not exposed to the end-user directly, for example, the hospital management system.

Refer to *Figure 1.3* that illustrates the 2-tier database architecture:

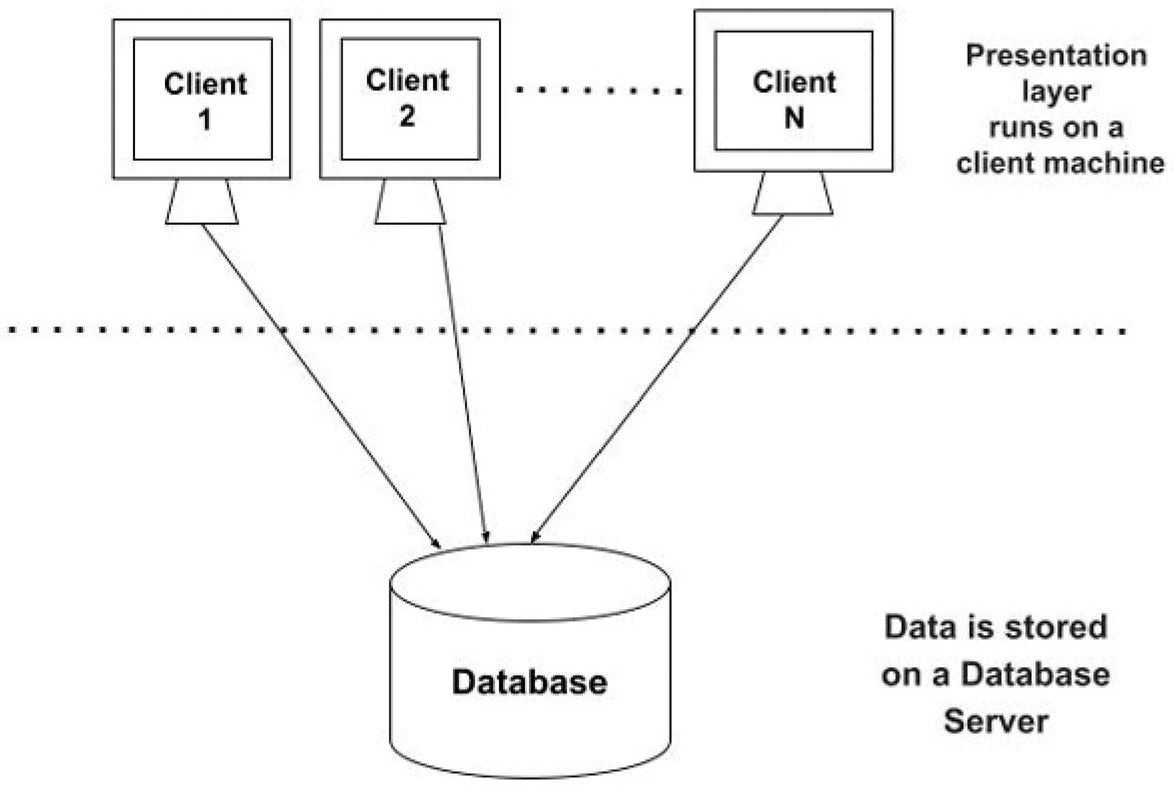

Client 1
Client 2
Client N
Presentation layer runs on a client machine
Database
Data is stored on a Database Server

Figure 1.3: *2-tier Architecture*

3-tier database architecture

Nowadays, DBMS has 3-tier architecture, which consists of the following:

- Presentation layer runs on a client machine
- Application layer (Business logic/server)
- Data is stored on a Database Server

In the 3-tier architecture, an additional presentation or GUI Layer is added, which provides a graphical user interface for the end-user. The application layer communicates with a database system to access data. The business logic decides the policy of the interaction and the nature of the actions for a given condition. Three-tier applications are more appropriate for large online applications.

Figure 1.4 shows the DBMS 3-tier architecture which divides the entire system into three independent and interrelated modules:

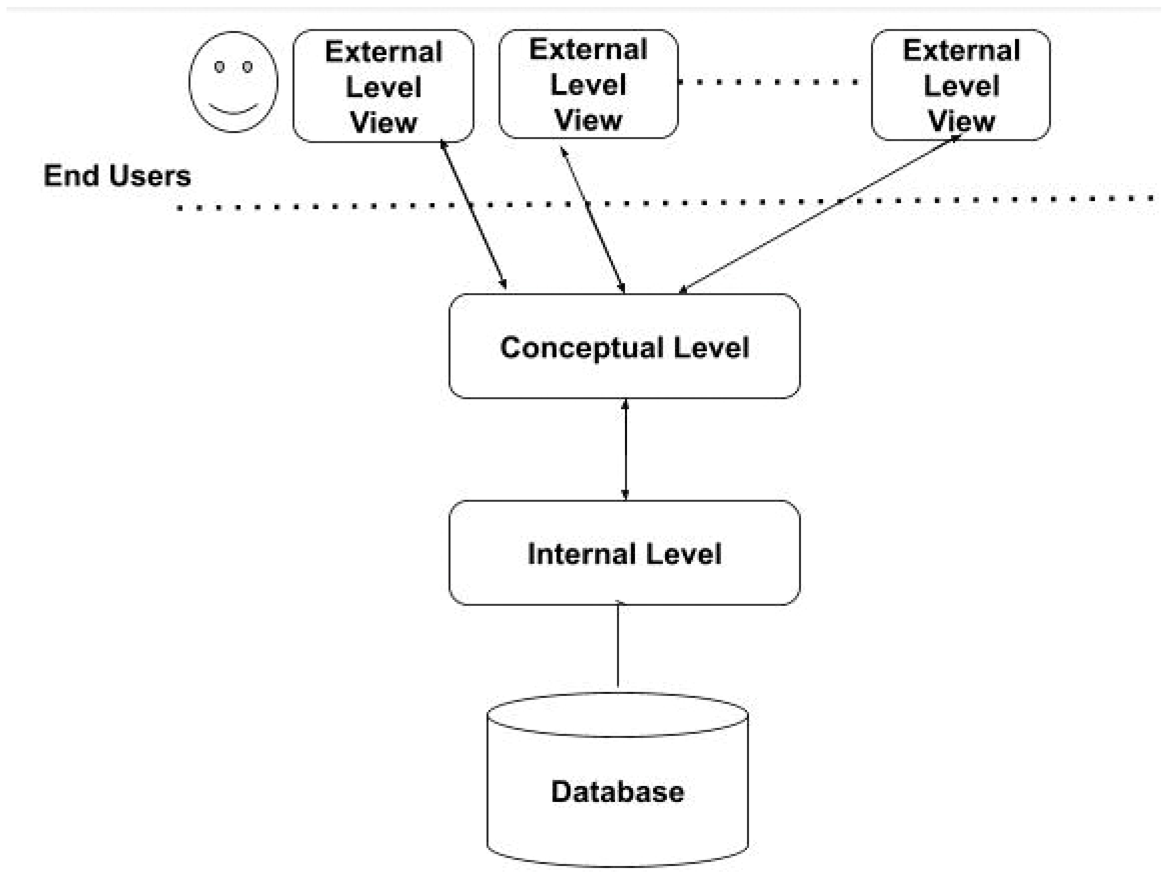


Figure 1.4: *Three-Schema Architecture*

Internal level

This is the lowest level of the Three-Schema Architecture. It is also known as the physical level. The internal level contains an internal schema that carries the details of the physical storage structure of the database. This level mainly focuses on how the data is actually stored in the database with complete details of data storage and access methods for the database.

Conceptual level

The conceptual level is present above the internal/physical level. It is also known as the logical level. The conceptual level contains a conceptual schema that carries the details about the structure of the whole database. This level mainly focuses on what data is to be stored in the database and how these data are interrelated.

For example, this level defines all the database entities, their attributes, and their relationships along with the security and integrity constraints.

At this level, the implementation details of the data structure are hidden.

External level

This is the highest level of the Three-Schema Architecture. It is also known as the view level. The external level has a number of external schemas. Each external schema (also known as sub-schema) shows part of the database to the end-users based on their requirements. End-user interaction with the database systems is possible at this level.

For example, at this level, TPO of the college may be interested to see the students' mark details for checking their eligibility for some company, whereas the accountants may be interested to check the fee status.

Data independence

It can be explained using the three-schema architecture, as shown in *Figure 1.4*. Data independence means changing the schema at one level without affecting the schema at the next higher level. Data independence helps us improve the quality of the data. According to its definition, it is broadly classified into the following two types:

- Physical data independence
- Logical data independence

Physical data independence

Making a change in the internal schema without affecting the conceptual schema is known as physical data independence.

For example, if we make any changes in the storage size of the database and it does not affect the conceptual schema, it means we have achieved physical data independence.

Logical data independence

Making a change in the conceptual schema without affecting the external schema is known as logical data independence.

For example, assume that TPO is interested in the students' marks. In this scenario, if DBA adds two more attributes (`Email, Project_title`) in the `Student` table, it will not affect the TPO's view of the table for the previous query. So, the data at the conceptual level schema and external level schema are independent and we have achieved logical data independence.

It is difficult to achieve logical data independence as compared to physical data independence as even minute changes done in a conceptual schema may be reflected in the user's view.

DBMS System Architecture

Refer to *Figure 1.5* that illustrates the DBMS system architecture:

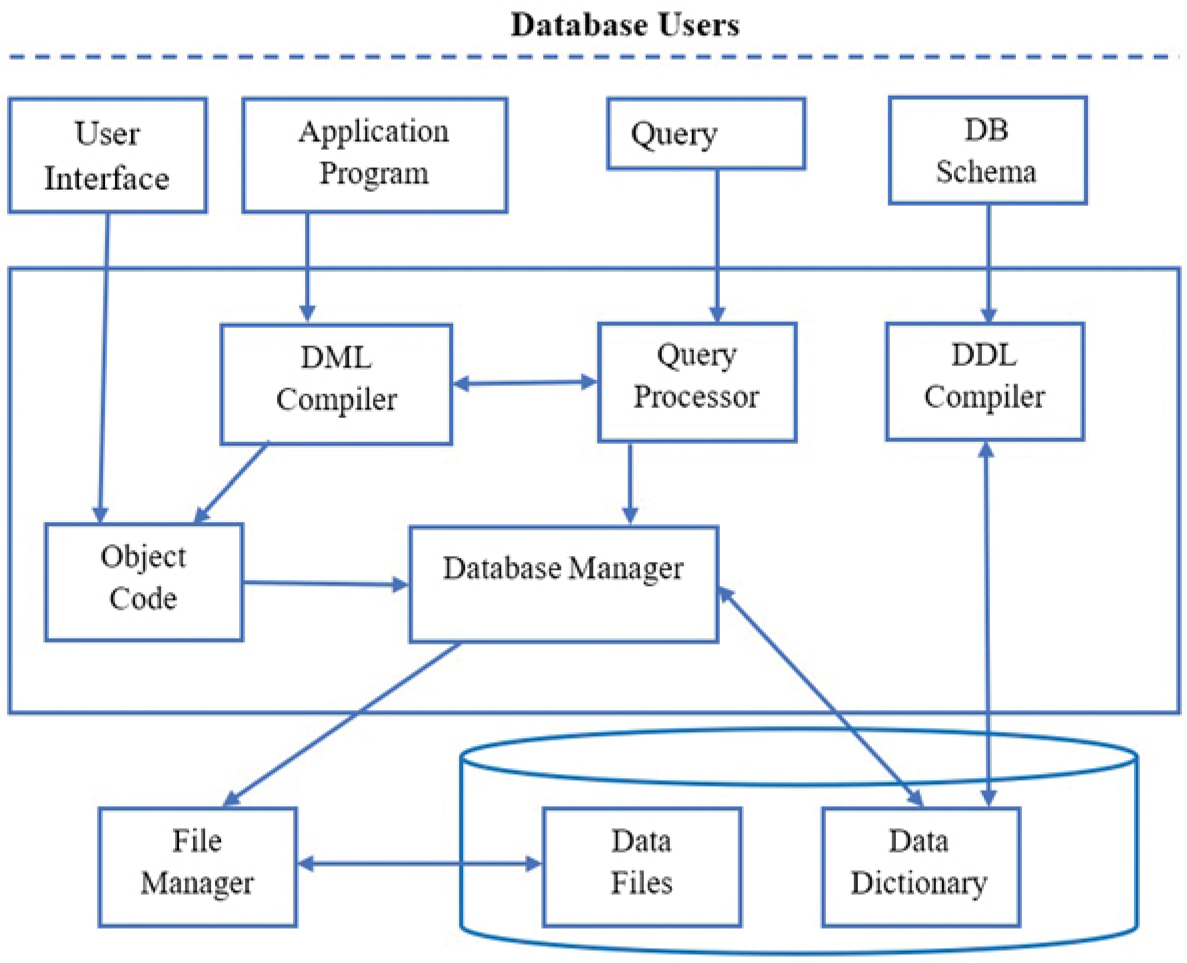


Figure 1.5: *DBMS System Architecture*

Figure 1.5 shows the major components of DBMS, which are the Data Definition Language (DDL), Data Manipulation Language (DML) Query processor, and the database manager. DDL is used to define the database structure. It inherently includes the data to be stored and structure to maintain the data. It maintains the relationship between the data to be stored as well as the constraints which it needs to be satisfied.

The DML is a set of tools used to process the data defined by the DDL, that is, DML is a set of programs or processes which work on the data. Even though it follows data processing, it does not have the traditional programming concepts as that of the other higher-level languages. The abstraction level makes it different. Data retrieval is much easier than traditional programming. For example, the sophisticated user may fire the query to find the set of students who secured first class. However, remember that it also has a procedural part.

The portion of DML used for retrieving the information is called a query language. Query language has a mathematical format and automatically processes the query given by the user. The query is not compiled to the object code but it is directly processed and the database is accessed. These functions are managed by the database manager module. It provides the security, concurrency, backup, recovery, and many more functionalities of the database system.

User interface, application program, query, and the database schema are the different inputs to the database system. The database schema defines the actual structure of the database. The application program may be written to access the data from the schema designed. Queries are the different questions by the users for data retrieval. User interface is the application interface for naïve users. It is a menu-driven program through which the naïve users can access the data from the database system. For example, in the banking system, the menu may be fixed for the debit and credit operations. These predefined menu-driven operations are used for accessing the data. In this case, withdrawal of the amount from an account or depositing money into an account. The naïve users (bank cashiers) can't write programs to access the data; however, they can make use of the available facilities (in this case, menu for the transactions).

Once the database schema is defined, the DDL compiler compiles it and the outcome will be the physical structure of the database being generated or created. The data will be stored in the data files and the actual data structure will be stored in the data dictionary. The data structure includes the details regarding the relations, fields, data types, etc., and are stored in the data dictionary. The application is written in the data manipulation language. The application program needs to be compiled which results in an object code. The executable object code written in DML does not have direct access to the details defined by DDL. The object code section of DBMS accesses the data and data structures by consulting the database manager.

The naïve user invokes the object code through the pre-written application programs to access the data. The query is a portion of data manipulation language, to retrieve the data through high-level calls. As we can figure out, the query input may not always get compiled. The query processor can generate the required commands based on the database scheme in association with the database manager. The query processing without compilation is an interesting feature of the database systems. Whereas, in the case of an application program, the DML compilation is required if the high-level language like Java fires an embedded query. The database manager uses the file manager which in turn returns the data stored in the data files at the operating/database system level.

Database Users

Database users are the persons who are using the database to fulfill their requirements. There are various types of database users and they can be recognized by the way they are working with the system, as depicted in *Figure 1.6*:

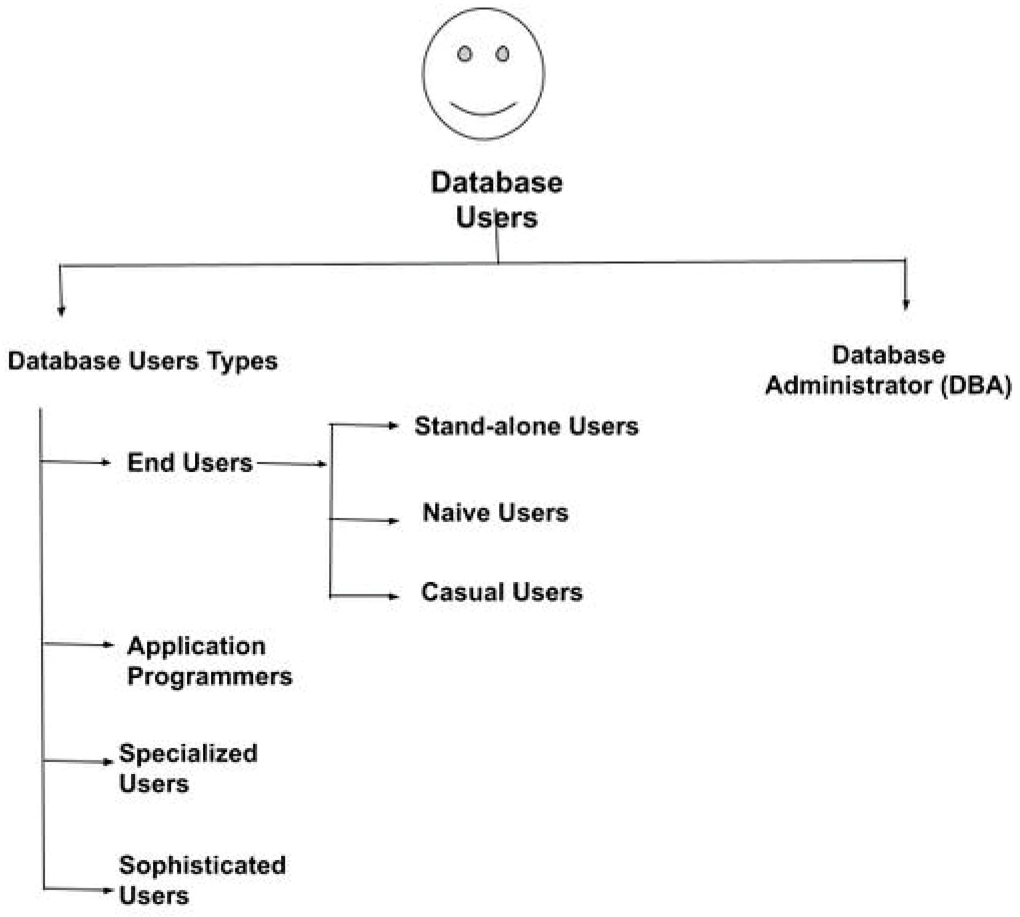


Figure 1.6: *Database users*

The following are the types of database users:

- **End users**: These users use the database applications or developed database applications and they don't have any knowledge about the design and complex details of the databases. End users can be again classified as stand-alone users, naive users, and casual users, and these are described as follows:
    - **Stand-alone users**: These are the users who use the databases for their personal usage. In the market, many database packages are available for such users, for example, tax calculator.
    - **Naive users**: These users are simple users who use the existing applications to communicate with the database. For example, when a passenger is using online railway reservation systems for

    booking a ticket, he/she creates the data in the database by entering and submitting his/her booking details into the system.

    - **Casual end users**: These users are rarely accessing the database but may require different information at different times. For example, the management accesses the data monthly to check the percentage of placement in a particular department.

- **Application programmers**: They are the computer professionals who write the application programs. These users interact with the database using DML queries. These DML queries are embedded in any application program, and these queries are converted into object code to communicate with the database. For example, writing a Hypertext Pre Processor (PHP) program to generate the report of students who are enrolling in a particular department. This will need a query to extract the data from the database. It will include an embedded SQL query in the PHP code.
- **Sophisticated users**: These users are not writing/using any application programs to interact with the database. In contrast, they write their own Structured Query Language (SQL) query to access the data. Designers and developers of DBMS and SQL can be considered in this category. Also, scientists, engineers, and analysts who conduct in-depth study of SQL and DBMS to apply their concepts to accomplish their needs.
- **Specialized users**: They are sophisticated users but they write special database application programs. Database developers come under this category. These users develop complex programs for a specific set of requirements. For example, rule-based systems, knowledge-based systems etc.

Database Administrator (DBA)

Database Administrator (DBA) is one of the database users and can issue many core operations within the database. DBA can be a single person or a group of persons based on the enterprise requirements. DBA has control over both the data and the applications. DBA is responsible for everything that is related to the database. He makes the policies and strategies and provides technical support. The roles and responsibilities of the DBA are as follows:

Schema definition

The logical definition or the overall structure of a database is nothing but the schema of the database. It is the responsibility of DBA to create the database schema. So, DBA issues multiple operations in DDL to define the schema of the database. Once the schema is defined, the users can perform various operations based on their role. If there are any changes in the operation of the enterprise or in the scope of the enterprise, then DBA is responsible to make the modifications in the database accordingly. In fact, it is very rare that the schema needs major changes as after designing, the DBA may forecast the future requirement and accordingly design the schema.

With schema definition, the other important tasks which are inherent to the schema definition are the storage structure and access methods definition. This will optimize the performance of storing and retrieving the data from the database.

Deciding data structures

Once the database contents have been decided, the DBA would normally make decisions regarding how the data is to be represented in the database, i.e, how the data is to be stored and what indexes need to be maintained. DBA must specify the representation by writing the storage structure definitions. In addition, a DBA normally monitors the performance of the DBMS and makes the changes to the data structures if the performance justifies them. In some cases, radical changes to the data structures may be called for.

Granting and authorization of users

Databases can be protected from unauthorized access by granting access based on the requirement and roles of the end-user. The task of granting the authorization of data access to different users is one of the tasks of DBA. The DBA may grant or revoke the access permission to the entire or parts of the database to the end-users.

For example, in an enterprise, the DBA may authorize every employee to view only his/her play slip. The end-user can’t change anything or can’t view the others' pay slips. The read-only access is defined by the DBA.

Routine maintenance

The database is an asset for an enterprise. From day-to-day working to a future plan, everything can be planned by using the database. So, DBA must ensure the database must be available to all the users 24/7. This requires the regular backup of the database. So, DBA is responsible for taking the database backup periodically in order to be able to recover the data from any failure due to human, hardware, or software malfunctioning and restore the database to a consistent state. DBA decides how frequently the backup needs to be taken or he will plan the schedule of foolproof backup.

New software installations and performance monitoring

The DBA is responsible for the installation of the new database software and/or application software and other related software based on the requirement of an enterprise. After the installation, he must ensure that all the operations, and especially the performance are as per the requirement of the enterprise. The DBA actually monitors the performance and usage based on the policies laid down by an enterprise. DBA monitors the CPU and memory usage and may take a call for performance tuning or new installation of the software.

Conclusion

Data is a raw fact which carries some inherent meaning associated with it. Processing the raw data will result in information. Database is a collection of interrelated data and DBMS is a collection of control routines to manage the database.

Maintaining data by using File Systems will result in data redundancy and data inconsistency. So, DBMS can be used to control redundancy and eliminate inconsistency.

DBMS provides security and concurrency control and maintains a data anomaly. DBMS provides an abstract view of data as well as data independence. Physical and logical data independence helps modify the schema at one level of the database system without altering the schema at the next higher level.

There are three different types of database architecture – 1-tier, 2-tier, and 3-tier. In 3-tier architecture, the business logic plays a key role in data access. End users, application programmers, sophisticated users, and DBA are the different types of database users. The main role of DBA is to design the database schema.

DDL, DML compiler, query processor, database manager, and file manager are the main components of DBMS architecture. DML achieves the data abstraction, whereas DDL is used to define the database structure. Query processor optimizes DDL and DML queries before processing it.

In the next chapter, we will discuss the conceptual data modeling introduced by *Dr. Peter Pin-Shan Chen*. It is commonly referred to as the **Entity Relationship** (**ER**) model. It shows the complete logical structure of a database.

Questions

1. List the database examples that you probably use in your daily life.
2. What are the disadvantages of storing organizational information in a file-processing system?
3. Explain how redundancy can cause anomalies in the database.
4. List the disadvantages of the file processing system. Explain how a database system can overcome it.
5. State the advantages of the database management system over the file processing system.
6. Differentiate between DBMS and file processing systems.
7. Give examples of the systems in which it may make sense to use traditional file processing instead of a database approach.
8. What are the disadvantages of a DBMS?
9. What is data redundancy, and which characteristics of the file system can lead to it?

10. What is meant by the data model? What different types of data models are available in DBMS? Explain these models.
11. Explain the difference between physical and logical data independence.
12. What is data independence, and why is it lacking in file systems?
13. Is Data independence one of the advantages of DBMS? Justify.
14. What is meant by data independence? Explain how it is achieved in DBMS.
15. What is the difference between physical data independence and logical data independence? Which is easier to achieve and why?
16. Describe three levels or layers of the DBMS system. How is this concept useful in physical and logical independence?
17. Explain in brief the different levels of data abstraction. Consider a two-dimensional integer array of size N x M that is to be used in your favorite programming language. Using an array as an example, illustrate the difference between the three levels of data abstraction.
18. Describe the overall database system with a neat diagram. Describe in detail its major functional components and major data structures used.
19. Explain the structure of the Query processor.
20. Define the following terms:

    data, database, Database Management System (DBMS), Relational Database Management System (RDBMS), Schema and subschema, Data Model, Database manager, DBA, End-user, Data independence
21. What different types of database users are available in DBMS? How do they interact with DBMS?
22. What are the responsibilities of the DBA ?
23. What are the different types of database end-users? Discuss the main activities of each user.
24. List the responsibilities of DBA. For each responsibility, explain the problems that would arise if the responsibility were not discharged.

CHAPTER 2
The Entity-Relationship Model

Dr. Peter Chen
(Born 1947)

Dr. Peter Pin-Shan Chen was born in 1947. He is a distinguished career scientist known for the development of the **Entity-Relationship** (**ER**) model in 1976. His original paper on the ER model is one of the most cited and most influential papers in the computer software field. It is commonly cited as the definitive reference for ER modeling. The ER model serves as the foundation of many system analysis and design methodologies. In database design and system development, the ER approach has been ranked as the top methodology. Dr. Peter Chen's work is a cornerstone of software engineering, especially the **Computer-Aided Software Engineering** (**CASE**) tools. He has received multiple awards including, but not limited to, the *Data Resource Management Technology Award* in 1990 and the *Outstanding Research Award* in 2004.

Introduction

A model is an abstract view. Modelling simplifies the problems by focusing on the important things and excluding the minute details. Data modelling is the first step in the process of database design, so is conceptual modelling for a successful database design. The ER model naturally represents the entities and their relationships of the real-world application. The ER model incorporates important semantic information about real-world applications. The model is based on set theory and relation theory. The main characteristic of an ER model is that it can achieve a high degree of data independence. The ER model can be used as a basis for a unified view of data.

Structure

In this chapter, we will cover the following topics:

- The main features of ER Model
- Defining relations between entities
- Impact of ER on database design
- Extended ER Model
- Real-world database design

Objectives

After studying this chapter, we will be able to understand the importance of conceptual models and define the basic concepts of the ER model. We will learn how to use Chen's ER notations for the real-world database design. We will also learn how to create an ER model for real-world applications. We will understand the limitations and the advantages of the ER Model.

Conceptual modeling of a database

What do you mean by "Model"?

A model is a template/prototype used as a sample to simulate the real-world systems. Generally, it is constructed for a better understanding of future solutions.

Why to make a “Model”?

It's a good idea to make models of your ideas before freezing the final solution. This will help you to determine whether your idea is going to work or needs modification.

A model allows you to test your solution quickly and cost-effectively. You could use the model for demonstrating to the clients and ask for design approval if design is what they are looking for?

Let’s assume that a car manufacturer designed and made a car without showing a model to the potential clients. If the car is found faulty, the company will have to face a huge loss.

That's why engineers build model airplanes, model homes/buildings, or model cars as a way to understand the reality of how a home/building or airplane or car is constructed. Engineers take feedback or discover new questions to ask, which helps make improvements.

“*Models help us to conceptualize an abstract idea more easily*”.

What do you mean by “Data Model” and why do we need to create it?

A data model is a collection of concepts that can be used to describe the schema of a database. The schema of a database defines what the entities are along with their data types, relationships, and constraints that needs to be applied to the data.

How to organize your wardrobe closet?

Your answer might be one of the following:

- Combine the storage units as per the needs – drawers for folded items, hangers for dresses and suits, and boxes for jewelry.
- Store most frequently used items at eye level, less-frequently used items below, and least-used items up high.

Likewise, one cannot store data in an unorganized format. Because once you create, gather, or start manipulating the data and files, they can quickly become disorganized. To save time and prevent errors later on, the organizations should decide how they will name and structure their data so that anyone can refer to it as and whenever it requires.

“ For these reasons, we need to create a Data Model” .

Database Model

Before we start to write any program or solve any mathematical questions or any assignment, we generally do some rough work to get the idea of the steps that will be required. In a layman’s language, data models are some concepts in which we mold our system beforehand. Doing so gives us a rough picture of how our actual system will look like.

A database model defines the logical schema of the data. The model describes the relationships between different parts of the data. For a database design, many models have been in use. They are listed as follows:

- Hierarchical model
- Network model
- ER model
- Relational model

A hierarchical database model is the oldest data model and uses a tree-like data structure to organize the data. The hierarchical model is useful to the data which has inherent hierarchies such as an organizational chart. It has a single parent and can have one or more children, that is, the parent:child has 1:N relationship. A network database model was formalized by the Database Task Group for flexible representation of objects and their relationships. It uses the graph data structure and can be visualized as a superset of the hierarchical model. In the late ’70's, hierarchical models and network models were used. But with the proliferation of ER model and relational model, it becomes almost absolute. So, let's discuss the ER model in-depth in this chapter.

Entity-Relationship Model

The Entity-Relationship (ER) model was introduced by *Chen* in 1976. It is a popular high-level conceptual data model. An ER model is a conceptual design of the database. In the ER model, the main components are entity, attribute, and relationship. For designing an ER Model, we will follow Chen's notations.

Let us describe the process for designing the ER models by illustrating all concepts of it through the college database example. Draw an ER diagram for the engineering college database.

Problem statement for engineering college database

Let us assume that the requirement gathering and analysis phase is already taken care of. And now, the database designer's job is to design the conceptual model for this application. So, the following are the detailed description of the engineering college database:

- **Department**: The college has different academic departments. Each department has a name, Head of the Department (HOD) who controls the department. It also keeps track of the start date when that staff began controlling the department. A department may have several laboratories and classrooms. A department offers a number of courses in each semester based on the syllabus scheme.
- **Staff**: The system needs to keep a track of all the staff details such as staff ID, name, salary, gender, address, contact number, date of birth (DOB), and date of joining (DOJ). Each faculty belongs to one department but may teach many courses. We also keep track of the task done by each staff (who is another staff member, for example, academic coordinator or internal quality audit coordinator of the department).
- **Student**: A student enrolls in a particular department and registers for various courses offered by the department. We need to keep track of the student name, registration number, gender, address, contact number, etc.
- **Course**: A department offers various courses in each semester. A course can be taught by one or more staff members. The students may

register for multiple courses. Each course has a course ID, name, and credits, etc. Each course may have prerequisite courses.

- **Parents**: The department maintains the parents' details of each student for sharing their academic performance and upcoming event details. The parents' information consists of name, email ID, address, contact number, relation with the student, gender, etc.

These are the major components of the engineering college database. We have excluded the other components and other minute details for the sake of simplicity. Now, let us discuss the main components of ER Model and will start with Entity.

The following are the features of an entity:

- An entity is a "thing" that can be distinctly identified.
- An entity represents something that exists in the real world and we want to maintain some data about them.
- A specific person, company, or event is an example of an entity.
- An entity can be the following:
    - A physical object - for example, House, Car, Player, etc.
    - A logical concept - for example, Job, Course, Server, etc.
- Entities are represented by labeled rectangles. The label is the name of the entity.
- Entity names should be singular nouns. And preferably represented by all characters in capital.
- For example, DOCTOR, TEACHER, PROJECT, etc.
- According to the *Engineering College Database*, STAFF, STUDENT, COURSE and DEPARTMENT are the examples of Entities, as shown in *Figure 2.1*:

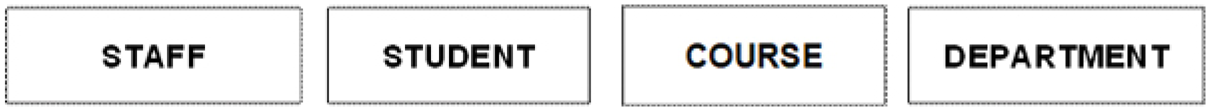
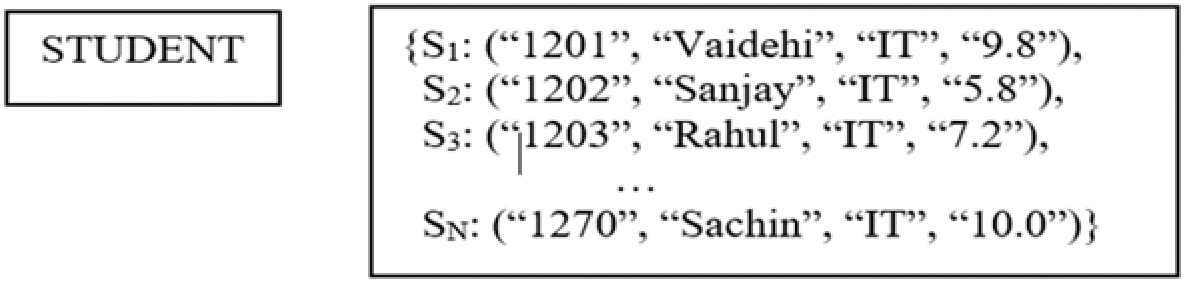


Figure 2.1: *Example of Entities in Engineering College Database*

The following are the features of an entity set:

- A database is modeled as a collection of entities and relationships among the entities. An entity is a physical object, or a logical thing described by attributes.
- An entity set is a collection of entities of the same type with the same set of attributes, i.e., set of all entities of the same type is called an entity set.
- In the ER model, every entity is represented with the name of the entity and set of attributes. For example, in the *Engineering College Database*, the entities are STAFF, STUDENT, COURSE, DEPARTMENT etc. The students are described by the set of attributes related to the STUDENT entity.
- Student Entity Set means the set of all students (S_1, S_2, S_3, S_4, …, S_N) enrolled in the department. All students enrolled in a department share the same set of attributes but have their own value(s) for each attribute, as shown in Figure 2.2:



STUDENT

{S_1: (“1201”, “Vaidehi”, “IT”, “9.8”),
S_2: (“1202”, “Sanjay”, “IT”, “5.8”),
S_3: (“1203”, “Rahul”, “IT”, “7.2”),
…
S_N: (“1270”, “Sachin”, “IT”, “10.0”)}

Figure 2.2: *STUDENT Entity and Entity Set*

The following are the features of attributes:

- An attribute is the set of properties, which are used to characterize the entity.
- Each entity has several attributes.
- Every attribute belongs to an entity and has a value. So, to know more about an entity, one can use the attributes.
- Attributes are represented by labeled Oval. The attributes are connected to entities with a line, as shown in Figure 2.3. The label is

the name of the attribute, preferably with the first capital letter.

- The name of an attribute must be self-explanatory.
- Attribute names should be adjectives.
- For example, in the *engineering college database*, one of the entities is STUDENT. It has attributes such as `Name, Reg_num, Address, Contact_number, Marks, Gender`, etc.
- Attributes give more information about the students. For instance, given the `Name` and `Reg_num`, one can find out the `Marks` of a student.

Refer to *Figure 2.3* that illustrates the attributes of student entity:

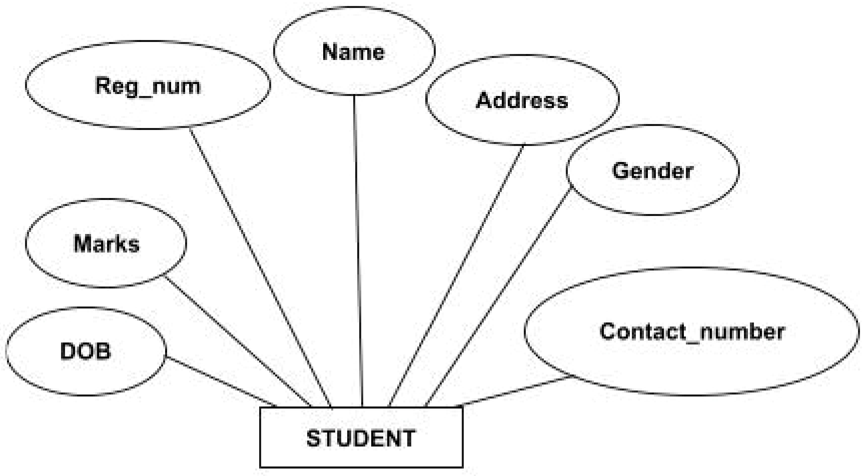


Figure 2.3: *The Attribute of STUDENT Entity*

Types of Attributes

Each attribute has a value that is stored in the database. Based on the value, we can classify the attributes as mandatory attributes or optional attributes. The mandatory attributes must have value, whereas the optional can be left empty. In a STUDENT entity, the Name attribute is mandatory as we must store the name of the student, whereas the phone value may or may not be stored based on whether the student has a contact number or not. Based on problem definition, DBA decides the required attributes in which the end-user is interested or would like to retrieve information.

Attributes can be classified as follows:

- Simple versus composite attributes
- Single-valued versus multi-valued attributes
- Stored vs. derived attributes
- Complex attributes
- Key attribute

I. Simple versus Composite attributes

Simple attribute

- Simple attribute consists of a single atomic value.
- A simple attribute cannot be subdivided further.
- For example, the Gender of a STUDENT entity is a simple attribute. It has atomic value as Male or Female which cannot be divided into subparts.

Composite attributes

- A composite attribute is an attribute that can be further split up without changing its meaning.
- A composite attribute can be subdivided further into sub-parts wherein every sub-part is atomic.
- For example, the `Name` attribute of a `STUDENT`. It can be further subdivided into first name, middle name, and last name as shown in Figure 2.4. Similarly, the Address attribute may be further subdivided into `Street, City, State`, and `PIN`. After division, each subpart of the composite attribute is a simple attribute, i.e., each subpart has an atomic value. Refer to Figure 2.4 that illustrates the composite attribute of student entity:

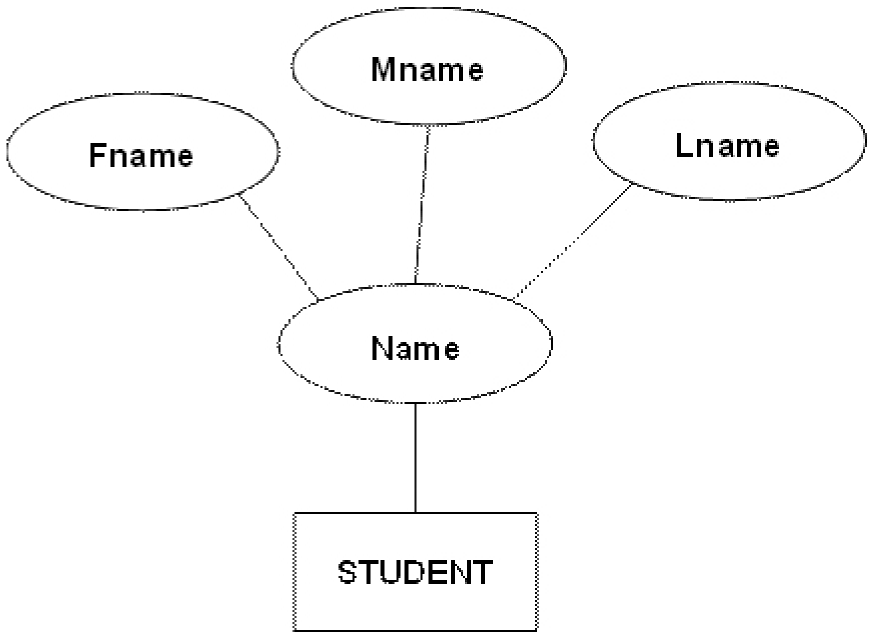


Figure 2.4: *Composite attribute of Student Entity*

II. Single-valued versus Multi-valued attributes

Single valued attributes

- A single-valued attribute can hold only a single value, for example, the `Reg_num` attribute or `Gender` attribute of a `STUDENT` entity has a single value.
- Single valued attribute can be a simple or composite attribute.
- For instance, `Date of birth (DOB)` is a composite attribute, '`Age`' is a simple attribute. But both are single-valued attributes.

Multivalued attributes

- Multivalued attributes can have multiple values.
- For example, the `Contact_number` attribute of `STUDENT` may have more than one value if a student has more than one contact number.
- Multi-valued attributes are represented by double oval, as shown in *Figure 2.5*:

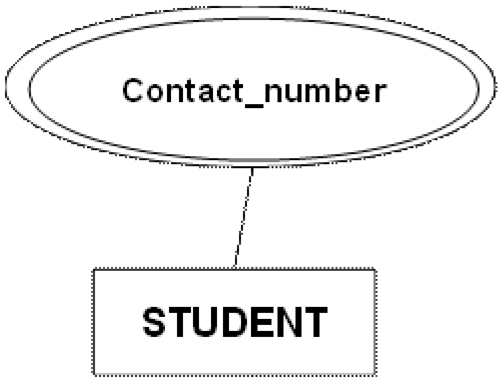


Figure 2.5: *Multi-valued Attribute of STUDENT Entity*

III. Stored verses Derived attributes

Stored attributes

- Some attributes are already stored in the database and can be used to derive the value of another attribute. Such attributes are called stored attributes.
- For example, in a *engineering college database*, `Marks` are a stored attribute and are used to compute the value of `Grade_pointer` of `STUDENT`.

Derived attributes

- Derived attribute value is derived from the stored attribute.
- Derived attributes need not be stored physically in the database.
- It is represented with a dotted oval.
- For example, as shown in *Figure 2.6*, '`Marks`' of a `STUDENT` is a stored attribute; the value for the attribute '`Grade_pointer`' can be derived from `Marks`:

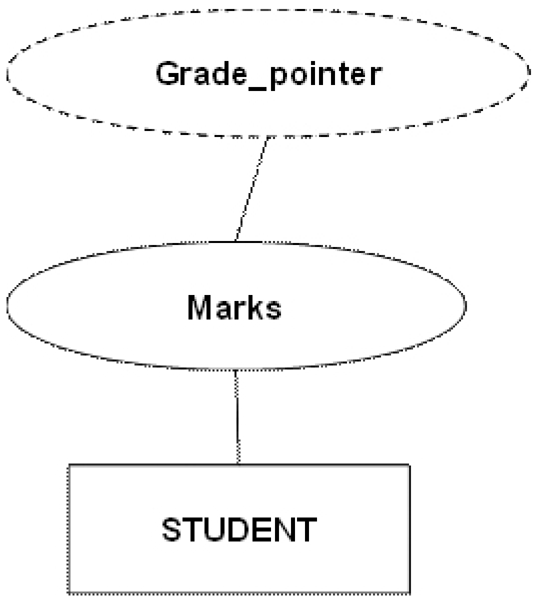


Figure 2.6: *Derived Attribute of STUDENT Entity*

IV. Complex Attribute

- If an attribute is both composite and multi-valued, then it is called a composite attribute.
- As it combines more than one form of attribute, it is called a complex attribute.
- For example, the address attribute of a STUDENT entity. We have already considered that the address is a composite attribute as it consists of more than one simple attribute, such as street, city, state, and pin code, etc. Now, if we assume that the student has more than one address, then it becomes a complex attribute. It is based on the requirement of an application and the design consideration, how the DBA models an attribute.

V. Key attribute

- A key attribute is the unique characteristic of the entity. An attribute is a key if the values of the attribute uniquely identify the instances of a corresponding entity set. Oval with underline labels represents the key attribute, as shown in *Figure 2.7*.
- Example. Student's registration number (`Reg_num`) is a key attribute of the entity `STUDENT` which helps to identify each student uniquely.

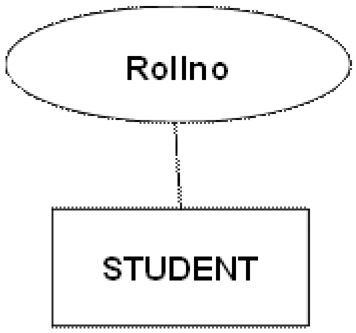


Figure 2.7: *Key Attribute of STUDENT Entity*

Relationship

- A relationship is an association between one or more entities.
- The entities participate in relationships. The relationship can be identified by the relationship name.
- For example, in the college database, a `STUDENT registers_in` for a `COURSE`. Here, `STUDENT` and `COURSE` are the participant entities and `registers_in` is the relationship name, as shown in *Figure 2.8*.
- Relationship types are represented using diamonds connected by lines to the entity types involved.
- Relationship names should be active or passive verbs, preferably with all the characters in small.
- Relationships between participating entities always operate in both directions. That means, while defining the relationship between the `STUDENT` and `COURSE` entities, specify the entity participation in the relationship. In the example of `register_in` relationship we can state the following:

  A `STUDENT` may register for more than one Course.

  A `COURSE` may be registered by many students.

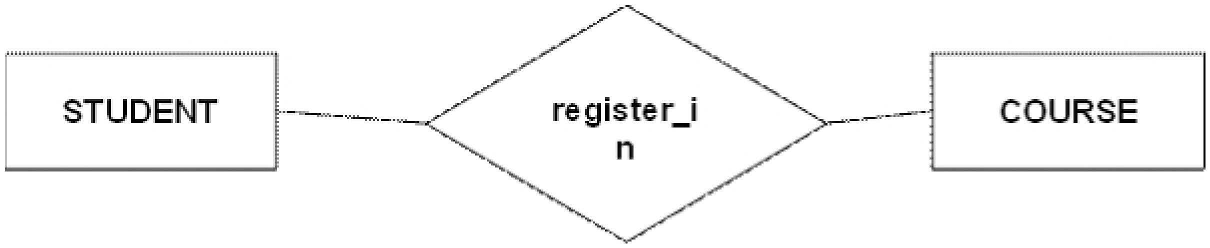


Figure 2.8: *Relationship between STUDENT and COURSE*

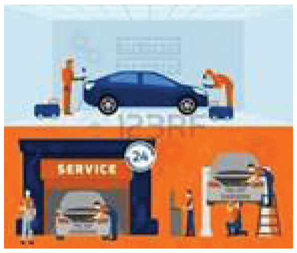
A few more examples to understand the relationship between entities are as follows:

"AUTHOR writes BOOK"

- Where AUTHOR and BOOK are the entities and writes is the relationship between these two entities.
- An author writes one or more books.
- A book can be written by one or more authors.

" TEACHER teaches STUDENT"

- Where TEACHER and STUDENT are the entities and teaches is the relationship between these two entities.
- A teacher may teach one or more students.
- A student may be taught by one or more teachers.

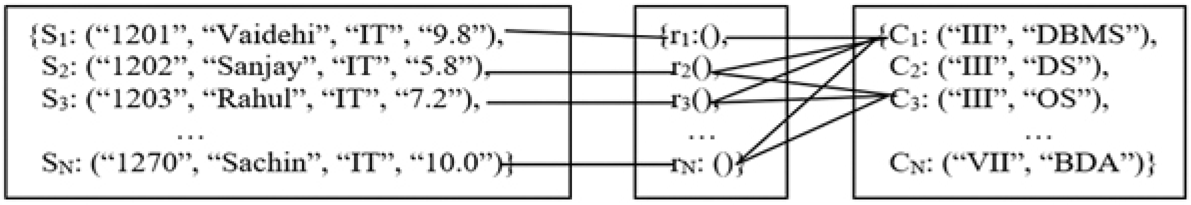


"MECHANIC repairs CAR"

- Where `MECHANIC` and `CAR` are the entities and repairs is the relationship between these two entities.
- A mechanic may repair one or more cars.
- A car may be repaired by one or more mechanics.

Relationship set

- The ER model represents the entities and associations for real-world application. It is the best practice to have a refined ER model before the actual implementation of databases for an enterprise.
- A relationship type defines a relationship set between the participating entities. For example, in the college database, the register-in relationship depicts the subset of cartesian products between the `STUDENT` and `COURSE` entities.
- Mathematically, the register-in relationship, R $\{r_1, r_2, r_3, \ldots\ldots r_N\}$ may be defined as a subset of the cartesian product of $\{S_1, S_2, S_3, \ldots, S_N\}$ with $\{C_1, C_2, C_3, \ldots, C_N\}$, as shown in *Figure 2.9*:


{S1: ("1201", "Vaidehi", "IT", "9.8"),
S2: ("1202", "Sanjay", "IT", "5.8"),
S3: ("1203", "Rahul", "IT", "7.2"),
...
SN: ("1270", "Sachin", "IT", "10.0")}
{r1:(),
r2()
r3()
...
rN: ()}
{C1: ("III", "DBMS"),
C2: ("III", "DS"),
C3: ("III", "OS"),
...
CN: ("VII", "BDA")}

STUDENT Entity Set **register_in Relationship set** **COURSE Entity Set**

Figure 2.9: *Register_in Relationship Set between STUDENT and COURSE Entities*

Relationship types and relationship constraints

Three types of relationships exist between entities, which are as follows:

- Binary relationship
- Recursive (also referred to as Unary relationship) relationship
- Ternary relationship

There are majorly two types of constraints on relationships, which are as follows:

- Cardinality mapping
- Participation (Total and Partial)

Let us understand, the relationship types and relationship constraints in detail.

Binary relationship

A binary relationship means the relation between two entities. For example, HOD controls the department, here two entities (`HOD` and `DEPARTMENT`) are involved in a relationship (controls), as shown in *Figure 2.10*:

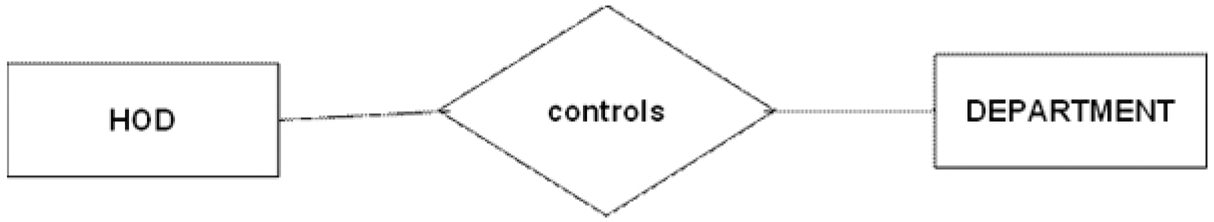


Figure 2.10: *Example of Binary Relationship*

Constraints on relationship: Cardinality mapping

An ER model is a visual representation of the database structure. It serves as a blueprint before the actual implementation of the database. The two main components of ER models are entity and relationships. It is better to know the mapping cardinalities of the entities in case of a relationship. Based on the participation, the cardinalities of the relationship can be classified into four categories.

The following are the four types of relationships:

- One to one mapping (1:1)
- One to many mapping (1:N)
- Many to one mapping (N:1)
- Many to many mapping (N:M)

1. One-to-one mapping relationship

When a single instance of an entity set is associated with a single instance of another entity set, it is referred to as one-to-one mapping cardinality. This type of relationship is rarely seen in the real world. Refer to *Figure 2.11* for the graphical representation

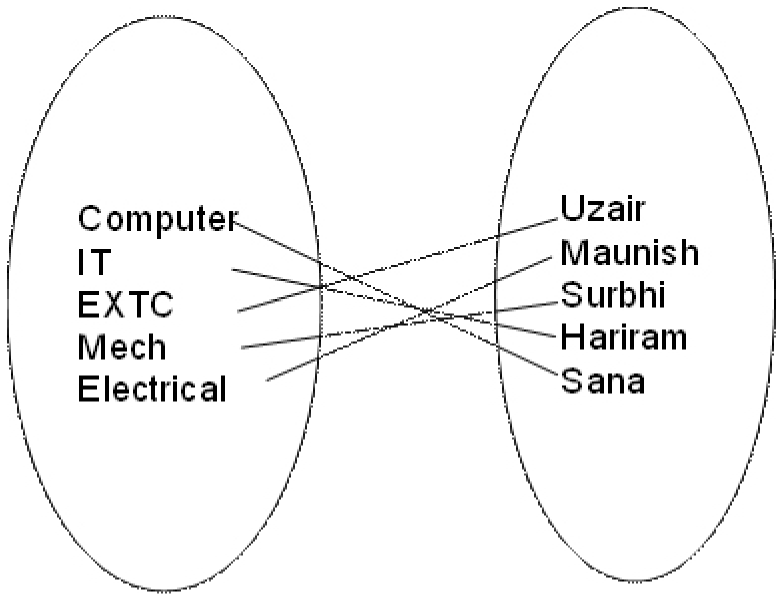


Figure 2.11: *Example of One-to-One mapping Cardinality*

For example, assume if `HOD` controls `DEPARTMENT` in college. That means, one department can have only one HOD and an HOD will also have only one department to control. Here, Sana is the HOD of the computer department, Hariram is the HOD of the IT Department, and so on, as shown in *Figure 2.11*. In the ER diagrams, the preceding relationship types can be represented, as shown in *Figure 2.12*:

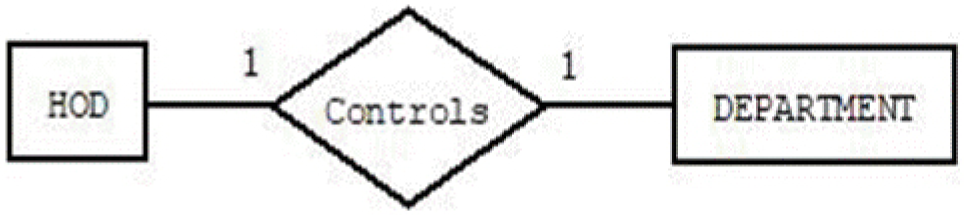


Figure 2.12: *Example of One-to-One Binary Relationship*

2. One-to-many mapping relationship

When a single instance of an entity set is associated with more than one instance of another entity set, it is referred to as one-to-many relationships. There are many real-life examples of the one-to-many relationships. A few are listed as follows:

- A customer can order many items from the store.
- A mother can have more than one child.

  Refer to *Figure 2.13* for a graphical representation of one-to-many mapping relationship:

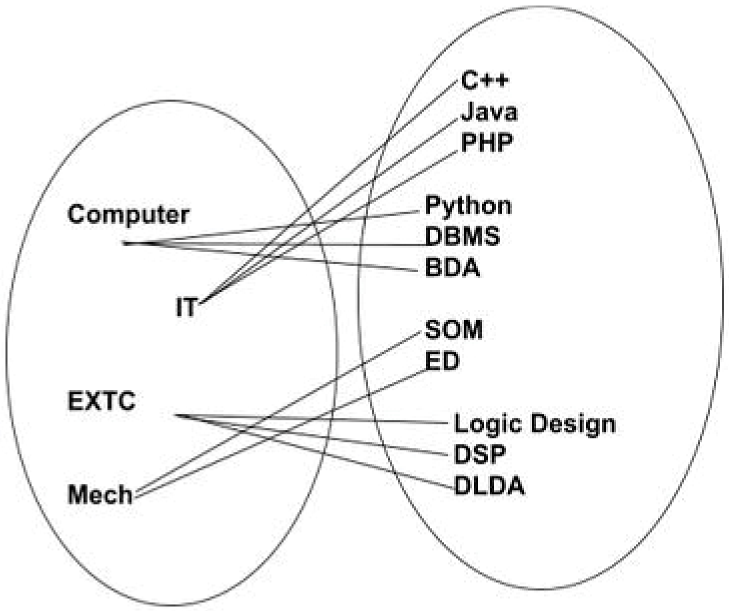


Figure 2.13: *Example of One-to-Many mapping Cardinality*

For example, in the *engineering college database*, a DEPARTMENT can offer many courses in a given semester. In *Figure 2.13*, one of the departments (Computer) offers many courses (Python, DBMS, BDA and so on). In the ER diagrams, the offers relationship types can be represented, as shown in *Figure 2.14*:

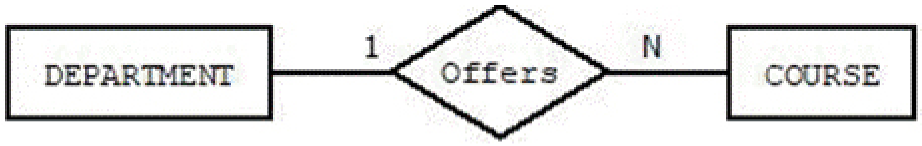


Figure 2.14: *Example of One-to-Many Binary Relationship*

3. Many-to-one mapping relationship

When more than one instance of an entity set is associated with a single instance of another entity set, then it is referred to as a many-to-one relationship. For example, in college databases, many students can study in a single DEPARTMENT, but a student cannot study in many

DEPARTMENTS at the same time. In other words, a student can register for only one program at a time, such as computer engineering or information technology. As shown in Figure 2.15, many students (Ammar, Sameer, Tina, Suman) enroll in one department (Computer), whereas few students (Ram, Aamir) enroll in another department (IT), and so on:

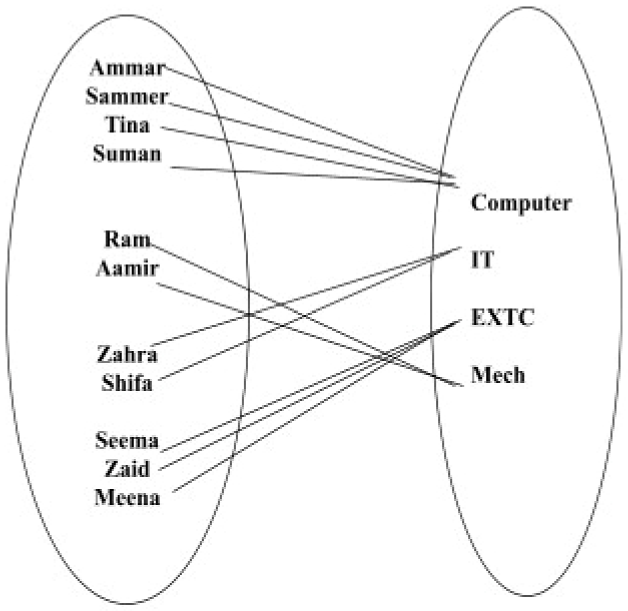


Figure 2.15: *Example of One-to-Many mapping Cardinality*

In ER diagrams, the preceding relationship types can be represented, as shown in *Figure 2.16*.

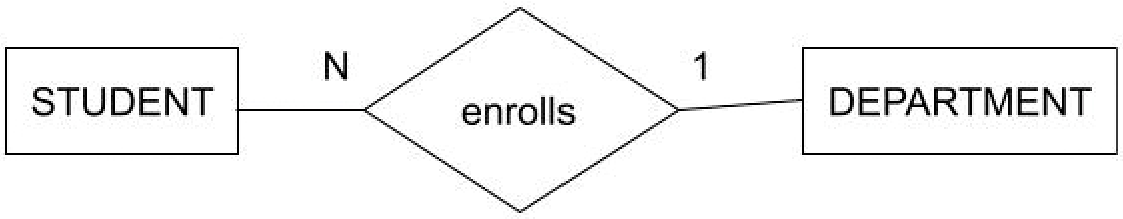


Figure 2.16: *Example of Many-to-One Relationship*

4. Many-to-many mapping relationship

When more than one instance of an entity set is associated with more than one instance of another entity set, then it is referred to as many-to-many relationships. For example, many staff may teach many courses. *Figure 2.17*, shows Sana and Hariram teaching many courses (DBMS, Python, and C++, Java, DBMS respectively) and DBMS subjects taught by many staff (Sana and Hariram):

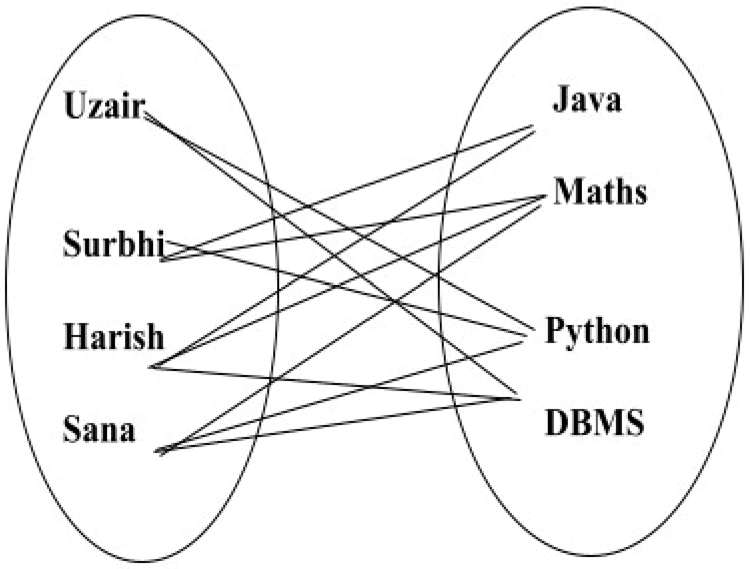


Figure 2.17: *Example of Many-to-Many mapping Cardinality*

In ER diagrams, the preceding relationship types can be represented, as shown in *Figure 2.18*:

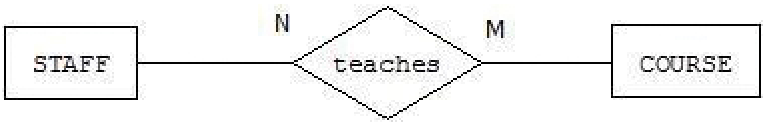


Figure 2.18: *Example of Many-to-Many Binary Relationships*

Another example from the *engineering college database*, the `register_in` relationship is many-to-many relationships. A student can register for more than one course, and a course can be registered by more than one student.

Constraints on relationship: Participation on relationships

The entities participate in relationships. This participation is optional or mandatory. As the relationship is bidirectional, it is important to decide the participation and connectivity of both entities. It is also important to decide the minimum and maximum entities participating in each direction in the relationship. That is, it is necessary to determine the cardinality of the participation. Based on this participation, one can decide whether it is optional or mandatory participation, as shown as follows:

- Total participation (Mandatory)
- Partial participation (Optional)

Total participation

When every entity from the entity set is participating in the relationship, then it is called total participation, i.e., in total participation, every entity instance will be connected through the relationship to another instance of the other participating entity types. For example, in the case of `STAFF` and `DEPARTMENT` entities and `works_in` relationship, every staff member works in department and every department must have staff, so it is total participation from both entities. In ER, total participation is represented as a double line between the participating entities to the relationship, as shown in *Figure 2.19*:

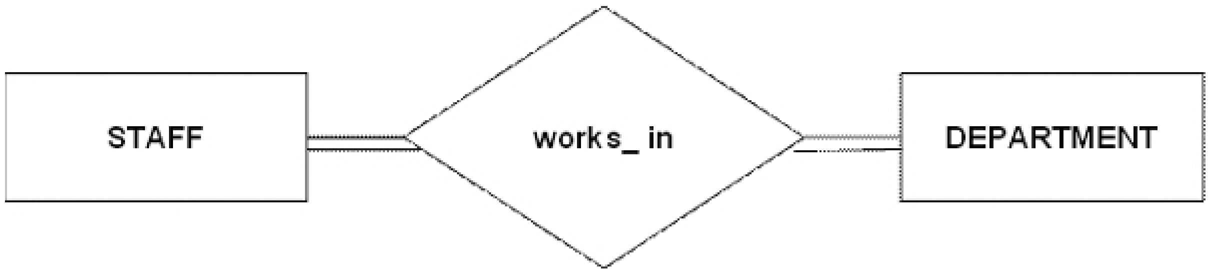


Figure 2.19: *Example of Total participation*

Partial participation

If only a part of the entity set is participating in a relationship, then it is called partial participation.

Consider the relationship – head of the department (HOD) from the *engineering college database*.

Here, all `STAFF` members will not be the head of the department. Only one `STAFF` will be the head for controlling the department. In other words, only one instance of the `STAFF` entity will participate in the preceding relationship and not all. So, the `STAFF` entity's participation is partial in the HOD relationship. But it is mandatory to have HOD for each department, so it is total participation from the `DEPARTMENT` entity, as shown in *Figure 2.20*; partial participation is represented by a single line in the relationship:

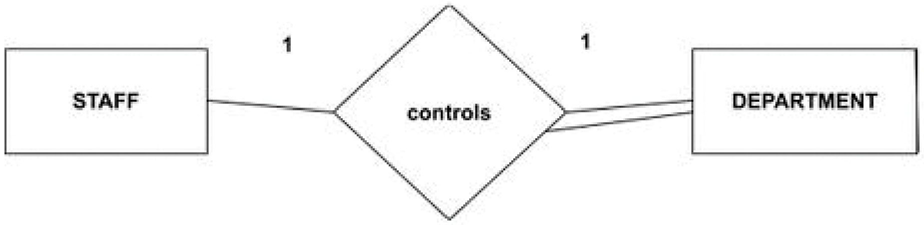


Figure 2.20: *Example of Partial participation*

Recursive relationship

- When an instance of any entity is related to instances of the same entity, it means an Entity is associated with itself, and such a relationship is known as recursive relationship.
- Recursive relationship is also known as reflexive relationship or unary relationship.
- For example, one staff may monitor the task of one or more other staff in a department.
- In the college database, there are two recursive relationships (`STAFF monitor_task STAFF` and `COURSE` prerequisite `COURSE`), as shown in *Figure 2.21*:

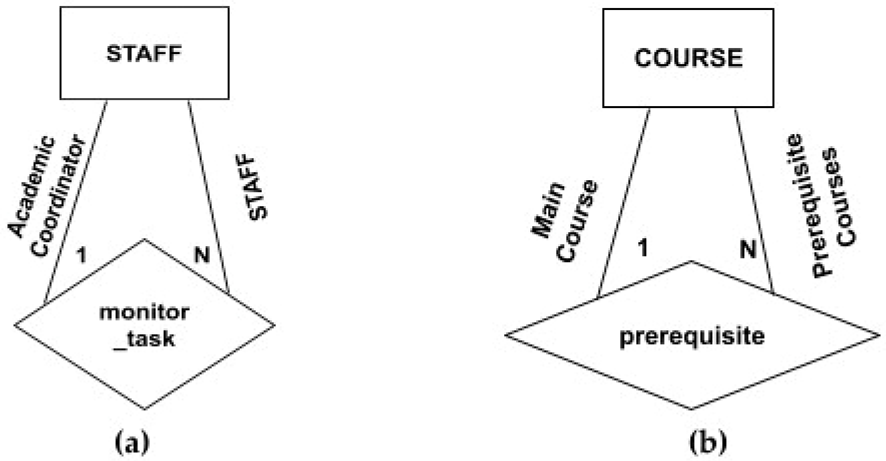


Figure 2.21: *(a) Example of Recursive Relationship (monitor_task) (b) Example of Recursive Relationship (prerequisite)*

Role names

- The functionality of the entity in the relationship can be explicitly represented by role names.
- As in recursive relationships, the same entity participates in more than one role, it can be represented explicitly as role names.
- For example, in *Figure 2.21(a)*, a STAFF can play the role of staff as well as an academic coordinator who monitors the task of the other staff.
- Another example from the *engineering college database*, which represents a recursive relationship is a course that may have one or more prerequisite courses, as shown in *Figure 2.21(b)*.

Attributes on relationship

- Entities have attributes that are used to describe entity. But relationships can also have attributes associated with them.
- Sometimes it is better to model the attribute as a part of a relationship instead of a single entity.
- For example, when any staff starts controlling the department, there is a need to capture the date of joining (DOJ) for the staff as an HOD. So, in this scenario, DOJ cannot be a part of the STAFF Entity and also can't be associated with the DEPARTMENT entity.
- If we model it as a part of STAFF, then for all staff members its value is NULL other than HOD. If we model it as a part of DEPARTMENT, then it will maintain the latest HOD's DOJ. This infers that DOJ has an equal role to play in the relationship for both entities. So, it is better to model it as an attribute of the relationship, as shown in *Figure 2.22*:

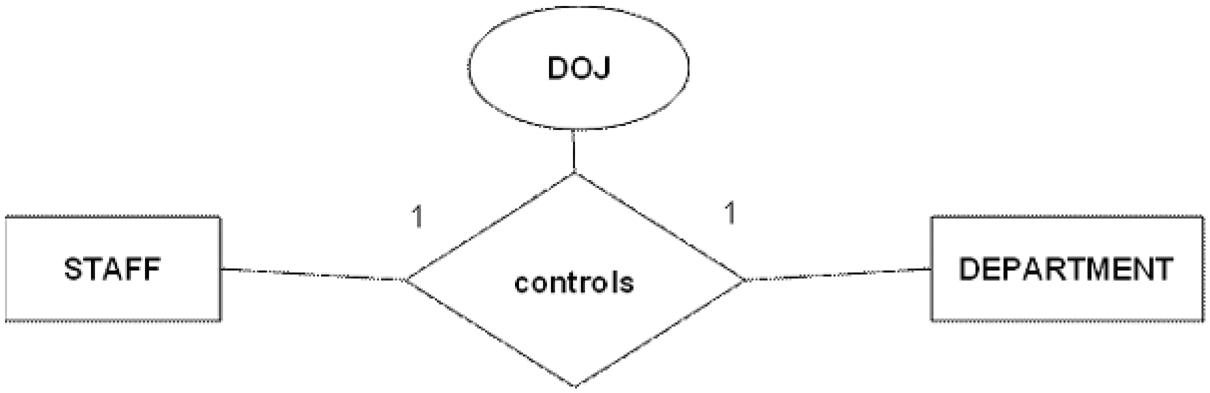


Figure 2.22: *Modelling Attribute as a part of Relationship*

Ternary relationship

The ternary relationship indicates that three entities participate in the single relationship or when three entities are associated in a relationship, it is referred to as the ternary relationship. When a binary relationship can't model a context, then a ternary or n-ary relationship can be used to describe the semantics of the relationship. As more and more entities participate in relationships, the modeling becomes more complex. The original paper of Chen describes the ternary relationship with the SUPPLIER-PROJECT-PART relationship set and is defined on three entity sets – SUPPLIER, PROJECT, and PART – as shown in *Figure 2.23*:

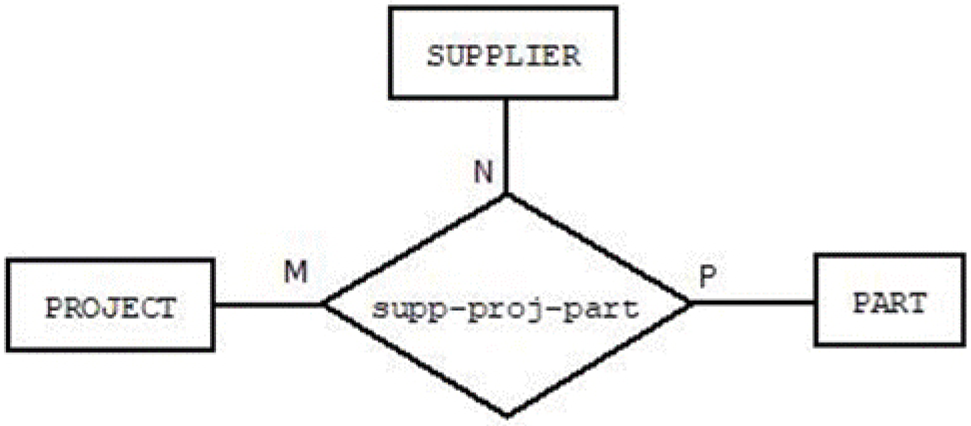


Figure 2.23: *Ternary relationship between SUPPLIER, PROJECT, and PART entities*

Figure 2.23 shows the ternary relationship between the `SUPPLIER`, `PROJECT`, and `PART` entities. The three entities are associated with a single `SUPP-PROJ-PART` relationship and the participation cardinality is also shown in Figure 2.24. The ternary relationship has three edges connecting each participating entity with the relationship. The SUPP-PROJ-PART relationship has many-to-many mapping cardinalities. It means more than one instance of the SUPPLIER entity set is associated with more than one instance of the PROJECT entity set and is also associated with more than one instance of the PART entity set. The three entities are associated with a single SUPP-PROJ-PART relationship and the participation cardinality is also shown in *Figure 2.24*.

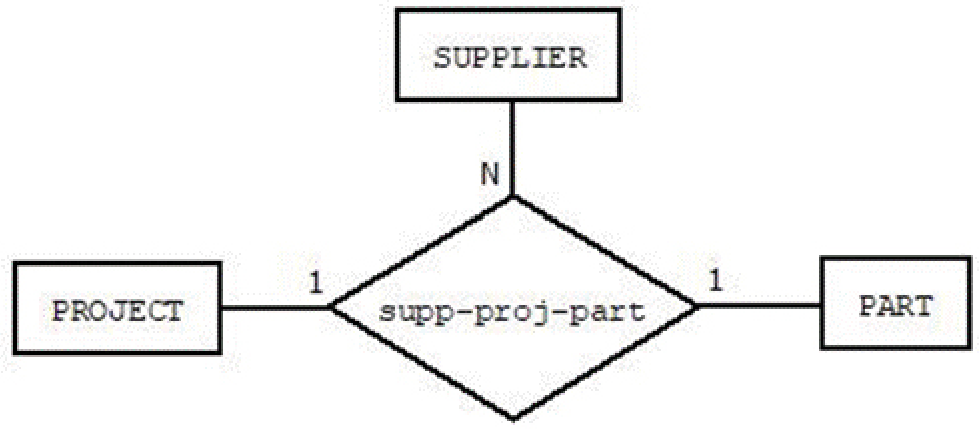


Figure 2.24: *Ternary relationship with different cardinalities*

Each supplier supplies a single part to a single project. It might be a specialty of a supplier to supply that part to a project. So, to describe a ternary relationship, we can consider a triplet, (S, P, T) which indicates that Supplier 'S' supplies, unique Part 'T' to Project 'P'.

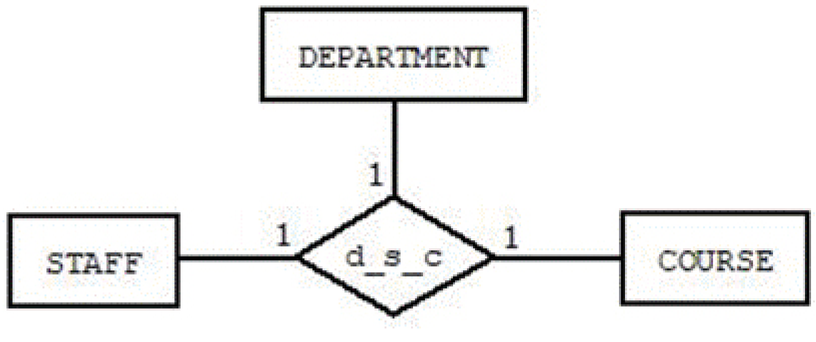


Figure 2.25: *Ternary relationship D-S-C of engineering college database*

In the *engineering college database*, we can consider the ternary relationship between the STAFF, DEPARTMENT, and COURSE, as shown in Figure 2.25. Here, the mapping cardinality is 1-1-1, which indicates that a single instance of STAFF entity belongs to a single instance of DEPARTMENT and teaches a single course from the COURSE entity, as shown in *Figure 2.26*:

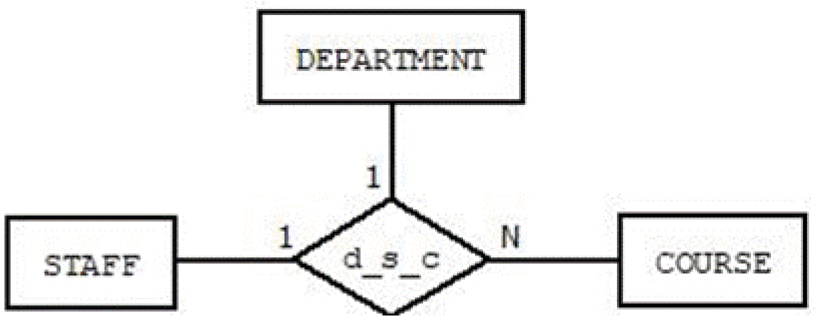


Figure 2.26: *Ternary relationship D-S-C of engineering college database*

Figure 2.26 shows different mapping cardinality for ternary relationship D-S-C of *engineering college database*. Here, the mapping cardinality is 1-1-N, which indicates that a single instance of STAFF entity belongs to a single instance of DEPARTMENT and teaches many courses from the COURSE entity.

The ternary relationship D-S-C shown in *Figure 2.26* associates DEPARTMENT, STAFF, and COURSE entities from the college database. The entities DEPARTMENT and COURSE are considered "many" based on the problem statement. That is, the department offers many courses in a given semester, and the entity STAFF is considered "one." This is represented by the following assertions:

Assertion 1: Single instance of the STAFF entity is associated with a single instance of a DEPARTMENT entity that could be teaching more than one course.

Assertion 2: A single instance of COURSE entity, offered by DEPARTMENT in a given semester, could be taught by more than one staff member.

Assertion 3: Each instance of STAFF entity belongs to one and only one instance of DEPARTMENT entity.

Weak Entity

The entities can be classified as strong entities or weak entities. A strong entity is one that has its own key attribute, i.e., the entity which is sufficient to form a key attribute is called a strong entity. In contrast to the strong, a weak entity is one that satisfies the following conditions:

- **Existential dependency**: The existence of an entity dependence on another entity which is a strong entity, i.e., it will be a part of the database only if the entity from the strong entity set participates in the relationship. The weak entity can't exist without the strong entity with which it has a relationship.
- The key attribute of a weak entity is partially or totally derived from the owner or identifying or parent entity in the relationship.
- The weak entity must have total participation in the relationship with the strong entity.
- Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner entity set, many weak entity sets).
- Weak entities are represented by labeled doubled rectangles in the ER model. The label is the name of the weak entity, preferably representing all characters in capital.
- For example, the `PARENT` entity in the *engineering college database*is a weak entity. The parents will be part of the database only if the student is part of the database. That is, the existence of `PARENT` depends completely on the `STUDENT` entity, so-referred to as a weak entity. Here, the parent can be Mother/Father/Guardian, as shown in *Figure 2.27*:

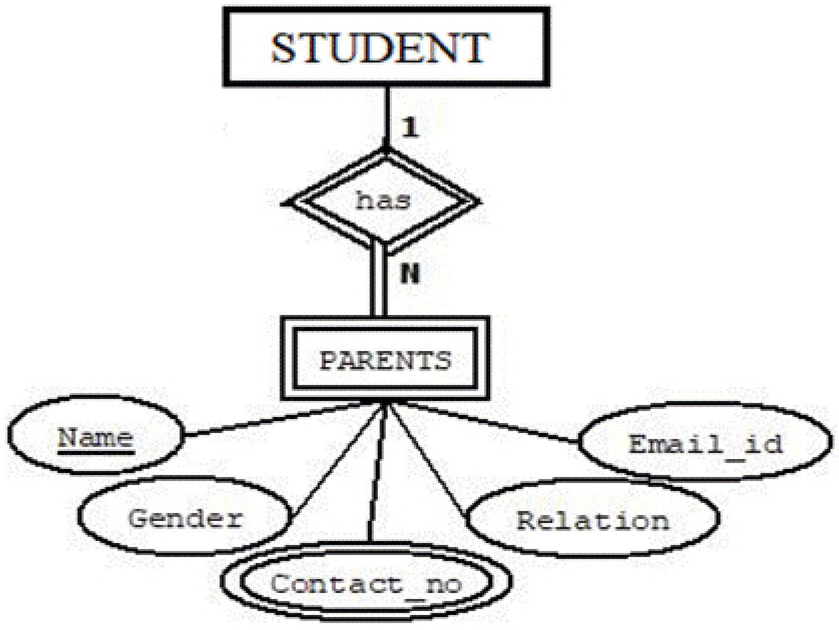


Figure 2.27: *Example of Weak Entity: Parent Entity*

Extended Entity-Relationship (EER) model

One of the limitations of the original ER model is that it did not support the generalization/specialization abstractions. But what is the need for generalization/specialization? As the complexity of the databases increases, it becomes difficult to model with the original ER model. Hence, there is a need to improve or extend the original ER model of Chen.

Generalization/Specialization in the ER Model is used for data abstraction. It avoids describing similar concepts more than once and makes the ER model more readable. This extended concept adds more semantic information to the design in an object form which is more acceptable than the traditional one. Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. In contrast, specialization is the process of defining subclasses based on some discriminating attribute. The subclass-superclass nature of EER is more realistic than the traditional approach.

Generalization

- Generalization is a bottom-up approach.

- When the common attributes from two or more lower-level entities are combined together and put in a top-level entity, the process is called generalization.
- For example, in the *engineering college database*, `STAFF` and `STUDENT` subclasses can be generalized into `PERSON` superclass by combining the common attributes from the subclasses (refer to *Figure 2.28*).

Specialization

- Specialization is a top-down approach.
- It is a top-down approach in which one higher-level entity can be broken down into two or more lower-level entities based on its distinct attributes; this process is called specialization.
- In the *engineering college database*, the `PERSON` is at a higher abstraction level, as shown in Figure 2.28. It has the common attributes of the `STAFF` and `STUDENT` entity. The specialized attributes become part of the `STAFF` and `STUDENT` entities which discriminate them into a specialized class.
- The `STAFF` entity can be further specialized into full-time `STAFF` or part-time `STAFF` or maybe a regular, ad-hoc, and visiting staff based on the requirement of an enterprise.

Constraints on specialization/generalization

There are two constraints on specialization, which are as follows:

- The disjointness and overlapping.
- The completeness and partial.

The disjointness and overlapping

- The 'd' symbol in the circle indicates the disjointness constraint. It means that the specialized entity can be a member of at least one of the specialized subclasses.
- When the specialization process is attribute defined and if the attribute is single-valued, then the specialized classes have disjointness constraint, otherwise it is overlapping.

- Overlapping constraint means after specialization, the entity may belong to more than one subclass. The ‘o’ symbol is used to indicate the overlapping constraint.

The completeness and partial

- Another important constraint on specialization/generalization is completeness/total or partial.
- As that of total (participation) constraint, the completeness or totalness constraint means the specialized subclass must be a member of at least one subclass.
- In case of partial constraint, the specialized subclass (entity) may not belong to any subclass.

In reality, the following types of constraints are possible on generalization/specialization:

- Total and disjoint
- Total and overlapping
- Partial and disjoint
- Partial and overlapping

CASE 1: Total and Disjoint types of constraint on Generalization/Specialization

As shown in *Figure 2.28*, a `PERSON` is involved in Total and Disjoint participation, which means a `PERSON` has been a member of either a `STAFF` class or `STUDENT` class but cannot be in both:

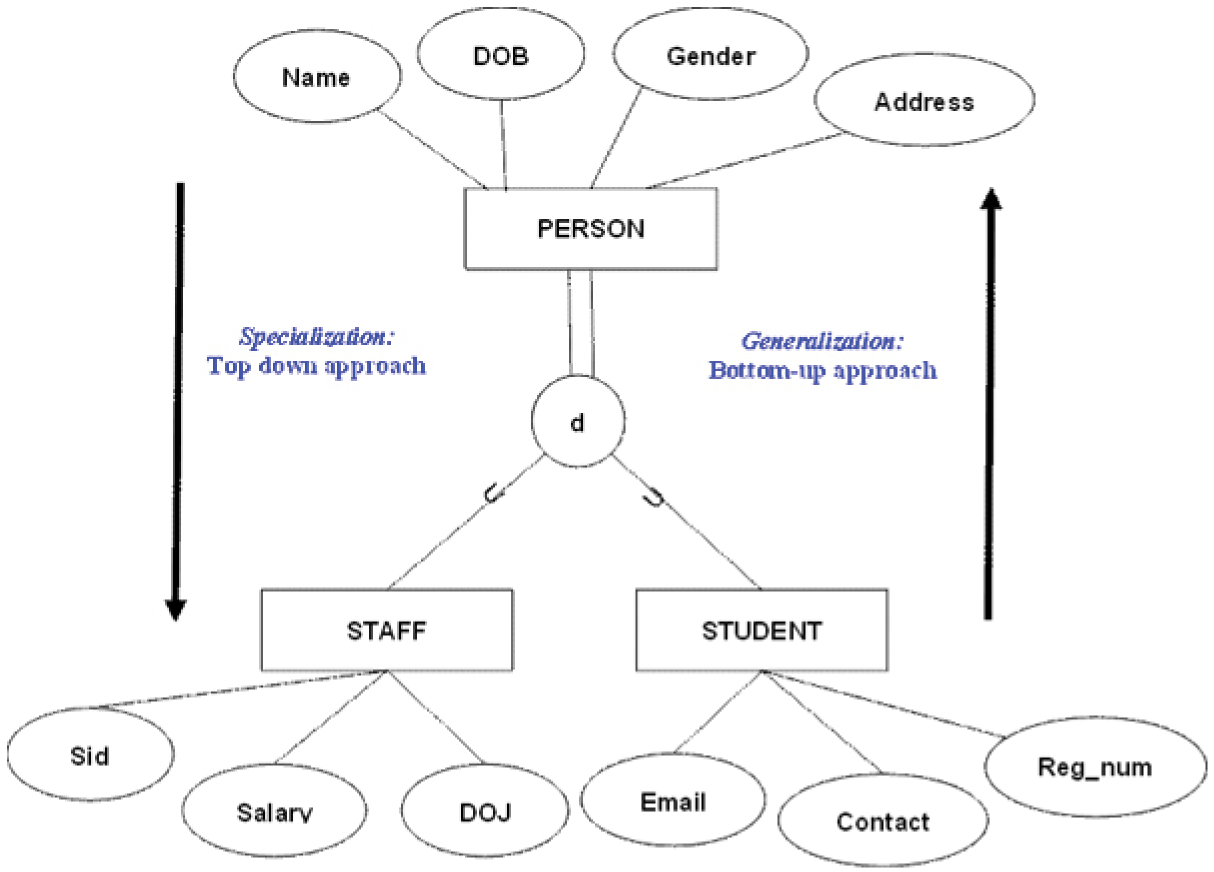


Figure 2.28: *Generalization (STAFF and STUDENT into PERSON) and Specialization (PERSON into STAFF and STUDENT)*

CASE 2: Partial and Overlapping types of constraint on generalization/specialization

As shown in *Figure 2.29*, a `STUDENT` is involved in Partial and Overlapping participation which means a `STUDENT` may or may not opt as `DEVELOPER` or `INTERN`, and also a `STUDENT` may have participated in both:

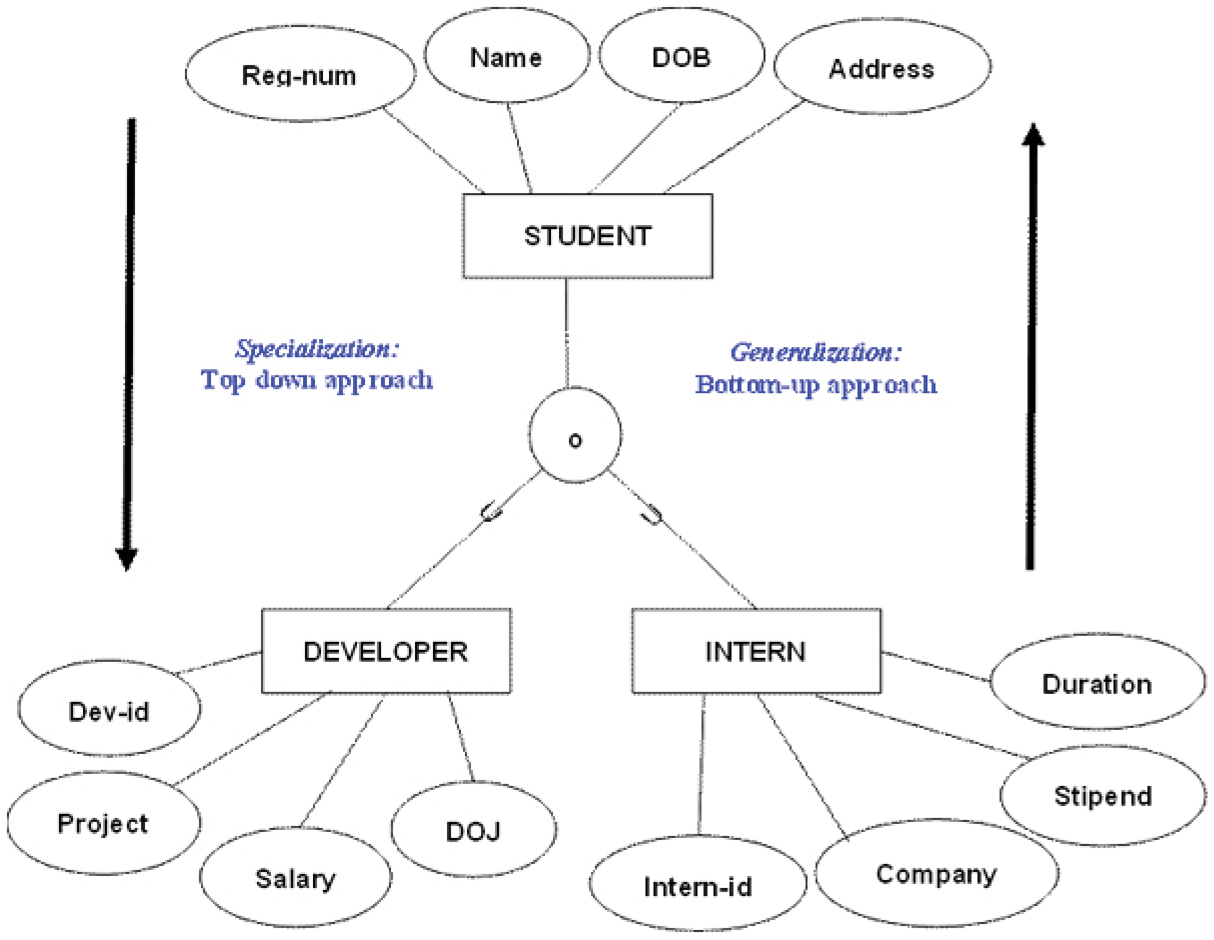


Figure 2.29: *Generalization (DEVELOPER and INTERN into STUDENT) and Specialization (STUDENT into DEVELOPER and INTERN)*

Aggregation

- The EER is used to overcome some of the limitations of the original ER model.
- Aggregation is an abstraction at a higher level.
- In an original ER model, a relationship is an association between one or more entities. That is, we can't model relationships among relationships. So, to model the relationship among relationships, we use Aggregation.
- For example, the department offers various courses in a semester. Students can register for the courses offered by the department provided s/he enrolled in the department. If a department offers a course, then only the students can register. That is, it is a relationship

between offers and `register_in` relationships, and can be modeled by using aggregation, as shown in *Figure 2.30*:

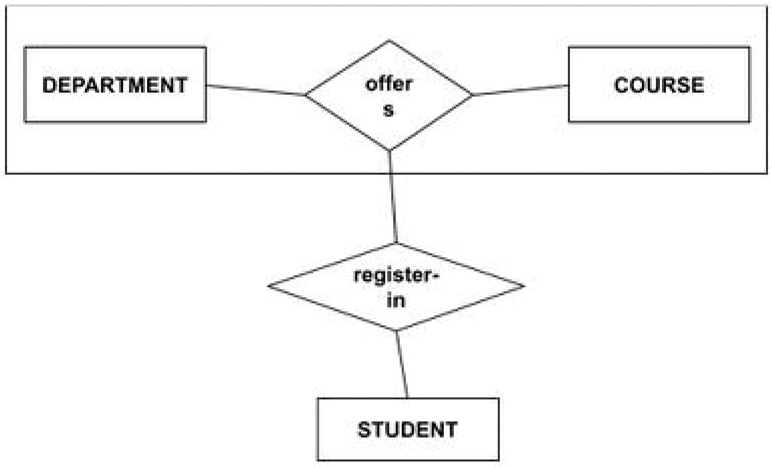


Figure 2.30: *Example of Aggregation in College Database*

Steps to create an ER diagram

Let's learn how to design ER diagrams with an example. (Refer problem statement for engineering college database mentioned under the section *Entity Relationship Model*). *Figure 2.31* shows the steps to create an Entity Relationship Diagram (ERD):

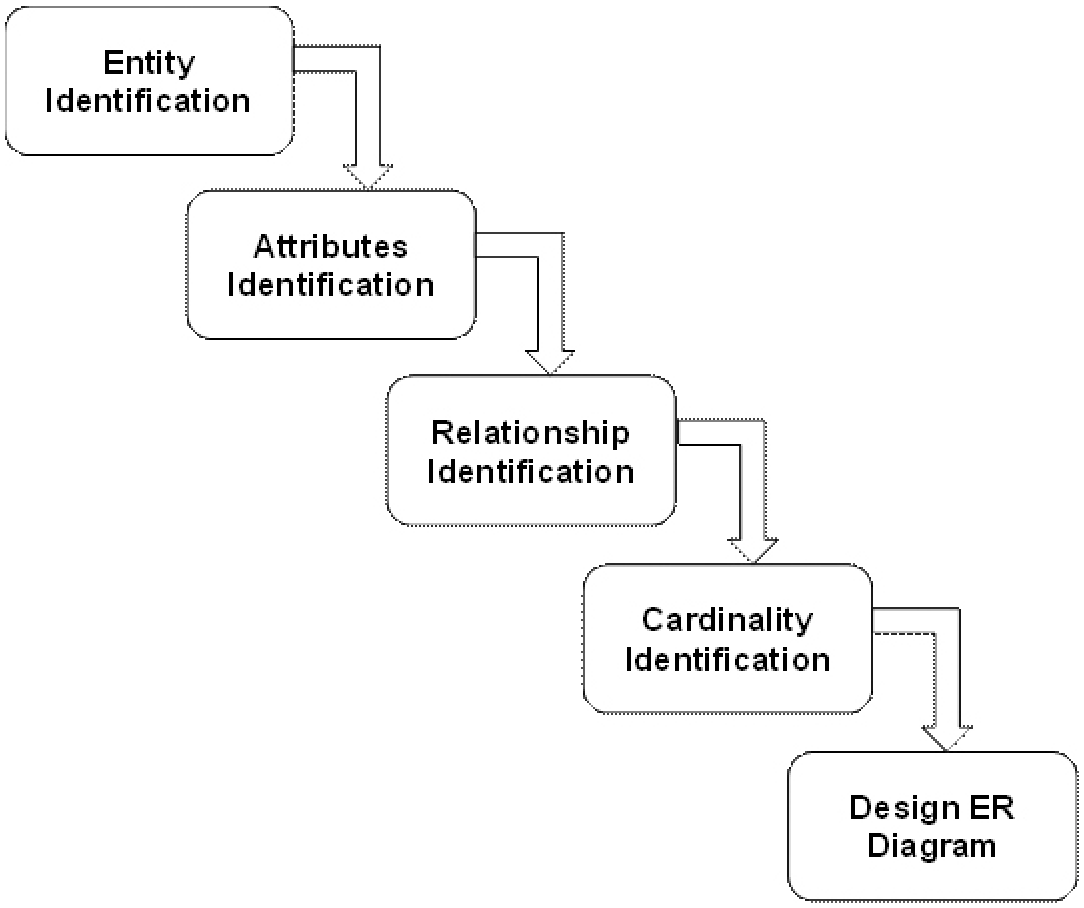

Entity Identification
Attributes Identification
Relationship Identification
Cardinality Identification
Design ER Diagram

Figure 2.31: *Steps for ERD*

Solution: ERD for engineering college database

Step 1 - Entity identification

- Strong Entity:
  `STAFF, STUDENT, DEPARTMENT, COURSE`
- Weak Entity:
  `PARENTS`

Step 2 - Attributes identification

- `STAFF:`
  <u>Sid</u>, `Name(Fname, Mname, Lname), DOB, DOJ, Gender, Salary, Address(State, City, Street, PIN)`
- `DEPARTMENT:`
  <u>Did</u>, `Name, Classroom, Labs`

- COURSE:
  <u>Cid</u>, Cname, Sem, Credits
- STUDENT:
  <u>Registration_number</u>, Name(Fname, Mname, Lname), Gender, Address(State, City, Street, PIN), Contact_number, Marks, Grade_pointer
- PARENTS:
  Name, Gender, Contact-no, Relation, Email-id

Step 3 - Relationship identification

- STAFF works_in DEPARTMENT
- STAFF controls DEPARTMENT (Attribute on Relationship controls: DOJ)
- STAFF monitor_task STAFF
- STAFF teaches COURSE
- STUDENT enrolls DEPARTMENT
- STUDENT register_in COURSE
- STUDENT has PARENTS
- DEPARTMENT offers COURSE

Step 4 - Cardinality identification

- STAFF works_in DEPARTMENT (N:1)
- STAFF controls DEPARTMENT (1:1)
- STAFF monitor_task STAFF (1:N)
- STAFF teaches COURSE (N:M)
- STUDENT enroll DEPARTMENT (N:1)
- STUDENT register_in COURSE (N:M)
- STUDENT has PARENTS (1:N)
- DEPARTMENT offers COURSE (1:N)

Step 5 - Design ER diagram

Refer to Figure 2.32 that illustrates the ER diagram for *engineering college database*:

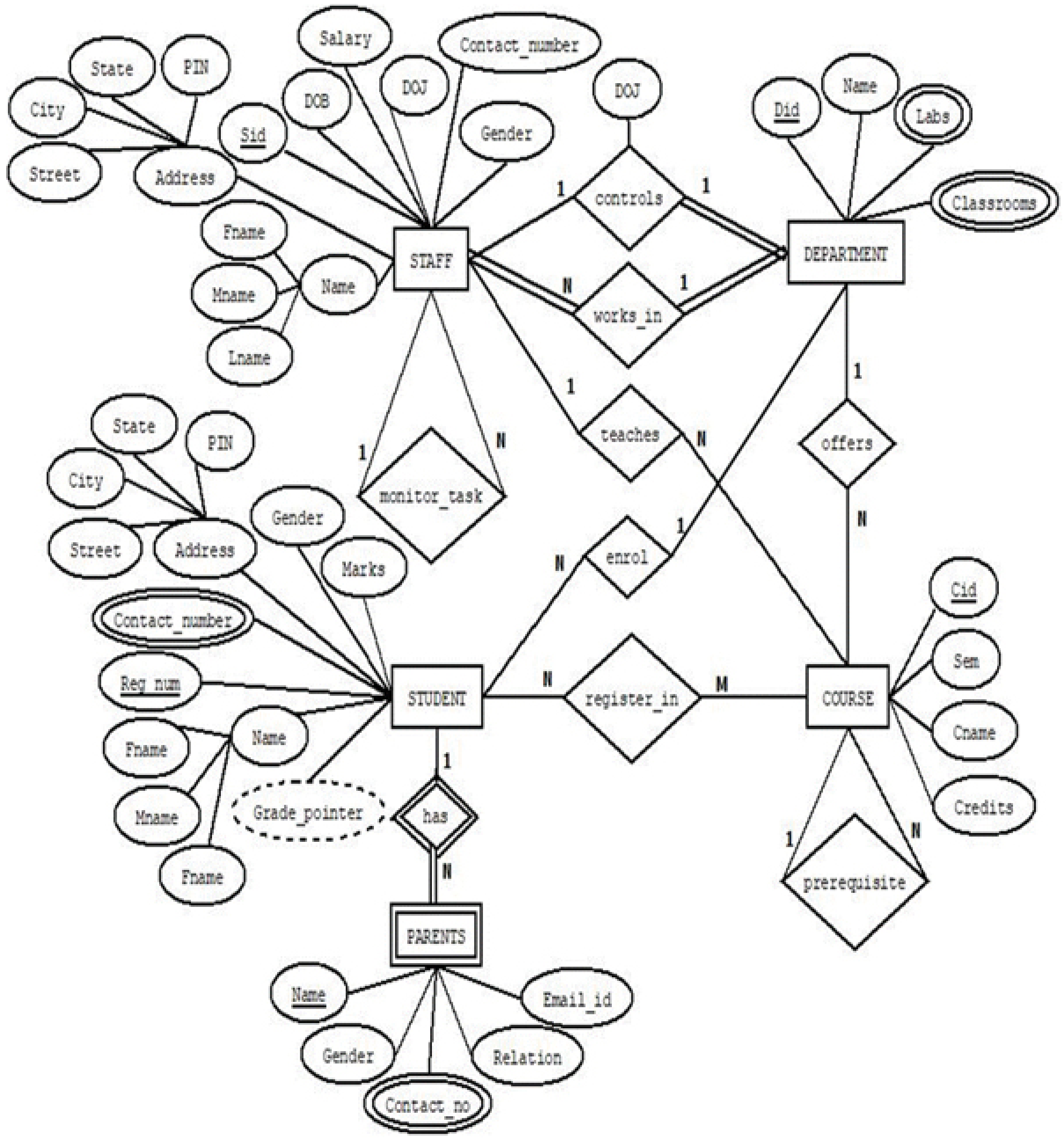


Figure 2.32: ER diagram for Engineering College Database

Conclusion

ER is a visual model used to represent and design the conceptual schema of a database.

The main components of ER model are attributes, entities, and relationships.

Attributes describe the entity. There are different types of attributes, such as simple and composite, single-valued and multi-valued, and stored and derived. Conceptual schema designer includes those attributes of an entity in which the end-user is interested to store and recall the information.

The key attribute is an attribute that uniquely identifies an entity from an entity set.

Entities are of two types – strong entity and weak entity. A weak or dependent entity is one that does not have a primary key.

A relationship is an association between entities. There are various types of relationships, unary or recursive, binary, ternary, and n-ary relationships.

The mapping cardinality of relationship represents the number of entities associated through a relationship set based on business rules and can be of 1:1, 1:N, and N:M types.

The specialization, generalization, and aggregation are the constructs of EER. The constraints of EER are participation which are of two types – total/partial and disjoint/overlapping. The user-defined predicate can be used for specialization. The specialization can be achieved using user-defined or predicate-defined subclasses.

A specialization is a top-down approach while generalization is a bottom-up approach. In the case of generalization, multiple lower-level entities are combined to form a single higher-level entity and contain generalized attributes entities. The higher level-entity will contain the common features or attributes.

Aggregation is a process when the relationship between two entities is treated as a single entity, i.e., using aggregation, we can express relationships among relationships.

In the next chapter, we will study the relational model. The relational model is based on the cornerstone research paper of *E. F. Codd*. The hierarchical or network models were used before the relational model, but the dawn of relational model became the dawn of database management systems as well. Since then, the relational database is dominating the database world.

Questions

1. What is the role of ER models in the database design process?
2. Define the terms entity and attribute. Explain different attribute types of ER models.
3. What is a strong entity and a weak entity? Give an example of each.
4. List the conditions to classify an entity as a weak entity.
5. What is a strong relationship and a weak relationship?
6. What is a participation role? When is it necessary to use role names in the description of relationship types?
7. Can we model relationships as attributes – Yes or No? If yes, justify with a suitable example.
8. What is a composite attribute? Explain with a suitable example.
9. Explain the terms primary key, candidate key, super key, and foreign key.
10. Differentiate between a strong entity set and weak entity set.
11. Discuss Extended ER features and the role of each feature in the designing of the database.
12. Define the concept of aggregation and give two examples where this concept is useful.
13. List the advantages of Aggregation.
14. Define the following terms:

    a) Superclass, Subclass

    b) Attribute defined specialization

    c) Mapping constraints
15. Use the ER model for the business rule: “An employee has more than one contact number”.
16. Discuss the constraints on specializations and generalizations.

17. Construct an ER diagram for the following database schema:

    person (Pid, name, address)

    car (license, year, model)

    accident (date, driver, damage-amt)

    owns (Pid, license)

    log (license, date, driver)

    The primary keys are underlined.

18. "*Vaidehi Software Incorporation*" is a software company, which has several employees working on different types of projects on different platforms. Projects have different schedules and they maybe in one of several phases. Each project has a project leader and team member at different levels. (Assume any other suitable information). Draw an ER diagram for Vaidehi Software Incorporation.

19. Construct an EER diagram for the hospital management system. It should include the patients admitted in the ward, doctors allocated to the ward, treatment/medicine given to the patient, and discharge of the patient. Assume the required attributes for the association among the entities and to maintain the integrity of the hospital data.

20. Draw an ER diagram for a Banking Enterprise assuming the Employee, Customer, Loan, Payment, Bank, Bank Branch, and Account are the major entities.

21. Draw an ER diagram for an engineering college which has various departments such as the administration department, Computer Engineering department, etc.

CHAPTER 3
Relational Model and Relational Algebra

Lawrence Joseph Ellison
(Born: August 17, 1944)

The research paper, "*A Relational Model of Data for Large Shared Data Banks*" (*E. F. Codd*, IBM Research Laboratory, Communications of the ACM, June 1970, pp. 377−387) became the cornerstone in the field of relational database design. When Codd proposed the relational model, most database systems were either hierarchical models or network models. It models the data in the form of relations. Its conceptual simplicity revolutionized the database design. It was a pioneering research project at IBM and UC-Berkeley in the mid-70s. Later, several vendors offered relational database products, including Oracle, and became a multibillion-dollar company.

Introduction

Ted Codd of IBM Research introduced the relational models in 1970 and still plays a vital role in the data processing applications. It stores the data in interrelated tables with unique names that represent the different types of entities and relationships between them. The simplicity and strong mathematical foundation eased the life of the users of the relational models. In this chapter, we will start with the fundamentals of relational models and gradually discuss the basic characteristics, keys, constraints, relational algebra, and relational calculus. The relational algebra, which is the basic building block for relational models, is discussed with its different operations and query processing. Further, it will cover the mapping from ER model (discussed in *Chapter 2, The Entity-Relationship Model*) to relational schema that can directly be implemented by any RDBMS. Many relational databases are based on the relational model; their straightforward way of representing the data in tables gives rise to many proprietary and open-source DBMS, such as MySQL, Oracle, and so on. It shows the simplicity, acceptance, and usability of the relational models.

Structure

In this chapter, we will cover the following topics:

- Logical view and data abstraction
- Basic components of relational model: relation, tuple, attribute

- Formal and informal definitions of relations and associated terms
- Relational database operators, data dictionary, and system catalog
- Handling data redundancy in the relational database model
- Relational Algebra basics and operations

Objective

After studying this chapter, we will be able to understand the fundamental concepts of Relational Models and define the different keys of the relational model. We will understand about integrity constraints and their importance in the database models. We will learn how to apply the different relational algebra operations and queries. We will also learn how to convert the ER model to relational schema and justify how data redundancy is handled in the relational database models.

Introduction to Relational Model

The Relational Model was proposed by *E.F. Codd* in 1970. It is simple, elegant, and based on the mathematical concept of relation. Informally, a relation is referred to as a table with rows and columns. Each row represents the record, and each column represents the attribute/field. This simple tabular representation of a relation is easy to understand for the end users as compared to the older data models. The main characteristics of relational model are:

- Relational model is used to store and process the data.
- Every relation has a unique name in the given database.
- A relation is a table with rows and columns.
- Each cell of the table/relation contains a single atomic value.
- Each attribute in the relation/table must have a unique name. But in the databases, the attributes with the same names may exist in different relations/tables.
- The order of attributes in relation does not matter.
- The order of tuples also does not matter or has no significance.
- It is based on the mathematical concept of relation.

Relational Model concepts

The COURSE entity, as shown in *Figure 3.1*, with conceptual representation becomes a relation in the relational model; we will discuss the mapping rules later in this chapter:

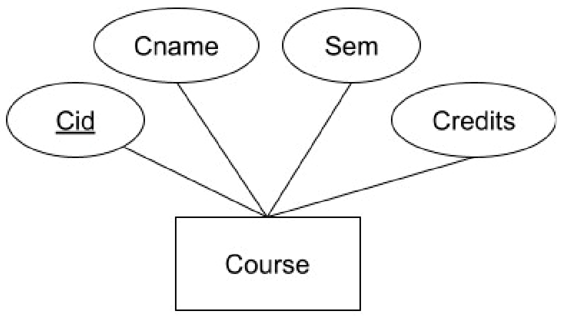


Figure 3.1: *Course Entity with attributes from Engineering College Database*

The entity becomes a relation, and the attributes are field names/column names of the relation. In the relation model, the COURSE is represented, as shown in *Table 3.1*:

Cid	Cname	Sem	Credits	Did
CC301	Database Management System	III	4	1
CC702	Big Data Analytics	VII	4	1
CC503	Web Technologies	V	3	1
CC304	Open-source Technology Lab	III	3	1
ITC301	Database Management System	III	4	2
ITC202	Operating System	IV	4	2
ITC705	Internet of Things	VII	3	2
EC303	Linear Integrated Circuit	III	4	3
EC502	Embedded System	V	3	3
MC304	Machine Design	III	4	4
MC603	Robotics	VI	3	4

Table 3.1: *Course relation - an instance of Engineering College Database*

Let us understand the important terminologies used in Relational Model, as follows:

- **Relation/Tables**: Informally, a relation is referred to as a table. The table has rows and columns. The row indicates the record of the entity, whereas the column shows the attribute of an entity.
- **Tuple**: Informally, it is the row of a table. A single row indicates the record of the relation.
  - The first row of the COURSE table shows the Database management system course of computer department..
- **Attribute**: It describes the entity which becomes the column in relation/table. It should be unique within the relation/table. The column represents the set of values for a specific relation. For example, Cid, Cname, Sem, and Credits are the attributes of the COURSE relation.

- **Degree**: Thetotal number of attributes in relation is referred to as the degree of the relation/table. For example, the degree of COURSE relation is five.
- **Cardinality**: The number of rows in a relation at a given instance is called the cardinality of the relation. For example, the cardinality of COURSE relation shown in *Table 3.1* is eleven.
- **Relation instance**: A relation instance is a set of finite tuples in a database at a given point in time. Any given relation/table does not have duplicate tuples. As we can observe from *Table 3.1*, the Cid attribute has a unique value for each tuple, and so it is called a key attribute.
- **Key attribute**: To avoid redundancy and inconsistency, each relation/table has an attribute or set of attributes which uniquely identify the tuple from the relation, is called a key attribute.
- **Domain**: Every attribute has a set of allowable values or range of values. This range is called the domain of an attribute.
    - For example, in case of EnggCollegeDatabase and COURSE relation, the attribute `Credits` can have a minimum value as 1 and a maximum value of may be 4. So, the domain values of `Credits` attribute are 1, 2, 3, and 4. The example for character/string-based domain are – Married (Yes or No) and SEX (M or F).

Relational model constraints and relational database schemas

So far, we have described the simple relation and its characteristics of single relations. A relational database is the collection of many interrelated relations. The tuples in those relations are related to each other in various ways. The state of attribute, the state of relations, and the relationship between various relations represents the overall state of a database.

The state of the database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or constraints on the actual values in a database state. These constraints are derived from the business rules of an enterprise which are required to be enforced. Some constraints must be enforced to maintain the consistency of the database, whereas some constraints are because of the integrity of the database. In this section, we will discuss the various constraints on the relational database design.

The relational database constraints can be broadly divided into three main categories, which are as follows:

A. Explicit constraints or schema-based constraints

Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the data definition language (DDL) are referred as schema-based constraints or explicit constraints.

- Data integrity means ensuring the quality requirements of the data in the database. If a database user tries to insert data that violates these requirements, DBMS will not allow it to execute.
- Whenever the database users perform any operations (such as insertion, deletion, or updation) on the database, these constraints will be checked by the system first; and if there is no violation in any of the constraints, the operation will get executed, else it will get rejected.
- There are many types of integrity constraints on Relational Models, which are as follows:
    - Domain integrity constraints
    - Entity integrity constraints
    - Referential integrity constraints

Domain integrity constraints

- These are attribute level constraints.
- Every attribute has a set of allowable values or range of values, which needs to be checked whenever the data is entered, is called the domain of the attribute.
- Relation schema defines the domain of every attribute/field in the relation. These are the domain constraints and needs to be checked at each instance of the relation.
- These business rule validations can be applied using Check and Not NULL constraints.
- Check constraints allow only a particular range of values. For example, if a constraint Sid>0 is applied on a `STUDENT` relation, inserting negative values of Sid will violate the rule and fail to execute.
- Not NULL constraints in a table ensures that the table contains values. When any column is defined as Not NULL, then that column becomes a mandatory column for the user, and s/he must provide the value for that column. For example, Not NULL constraint on the Name of `STAFF` relation implies that a value must be entered for the Name column.
- For example, in case of the *Engineering College Database*and COURSE relation, the attribute Credits can have a minimum value as 1 and a maximum value as may be 4. So, the domain values of the Credits attribute are 1, 2, 3, and 4, as shown in *Figure 3.2*:

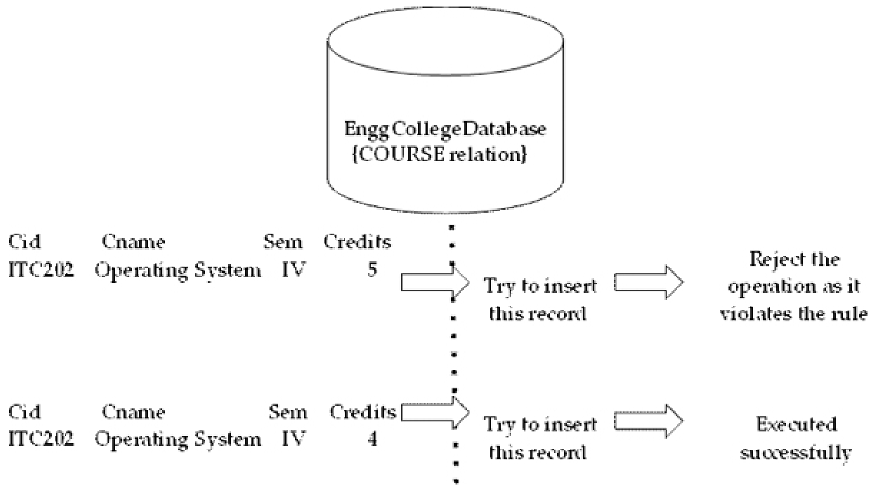

Engg College Database
{COURSE relation}
Cid Cname Sem Credits
ITC202 Operating System IV 5
Try to insert this record
Reject the operation as it violates the rule
Cid Cname Sem Credits
ITC202 Operating System IV 4
Try to insert this record
Executed successfully

Figure 3.2: *Example of Domain Integrity Constraints*

1. Entity integrity constraints

- The key attribute in a relation is used to uniquely identify each row. This requirement is called the entity integrity constraint.
- The business rule validations can be applied on a column using a primary key or a unique key constraint.
- Unique key:
  - The unique constraint can be applied on a single column or a group of columns as a unique key.
  - This allows only the unique value to be stored in the column. It rejects duplication.

- A table may have many unique keys.
- For example, in the *Engineering College Database*, the Department relation has attributes, Did and Dname. As the department names are unique in the database, we can apply a unique constraint on the Dname column of the DEPARTMENT relation.

- Primary Key:
  - Primary key is a column or group of columns in a table that uniquely identifies the rows in a table.
  - Its value should be unique for all the tuples and also can't have NULL values.
  - A table can have at the most one primary key.
  - For example, in the *Engineering College Database*, in the DEPARTMENT relation, the Did attribute is used to uniquely identify each department, hence one can apply the primary key constraint on it.

Refer to *Figure 3.3* that illustrates the example of Entity Integrity Contraints:

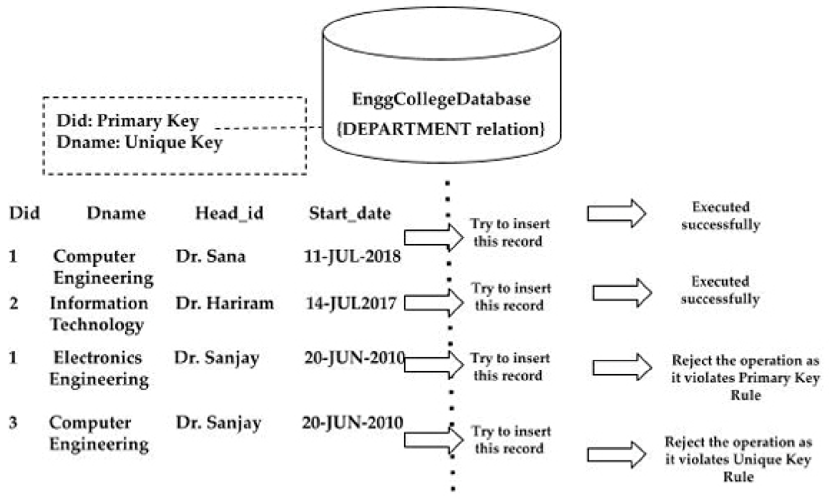


Figure 3.3: *Example of Entity Integrity Constraints*

2. Referential integrity constraints

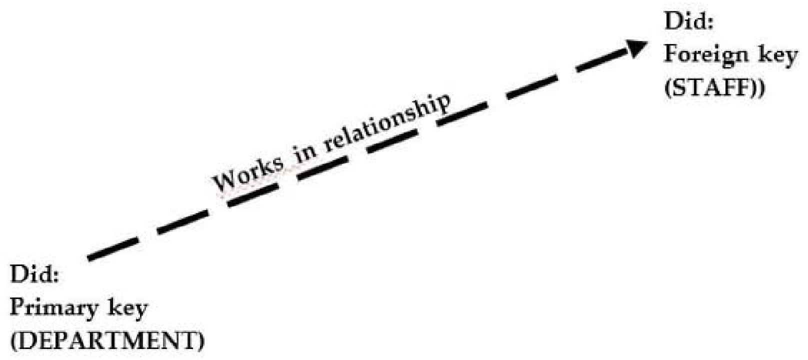
- Referential constraints enforce the relationship between the relations. When we want to ensure that one or more sets of attributes of a relation are related to one or more set of attributes in another relation, it is called referential integrity.
- These business rule validations can be applied on a column using the primary key and foreign key constraints, where the relation referring to a key attribute (Primary key) of a different or same relation becomes the foreign key.
- **Foreign key:**
  - The foreign key can be a column or a set of columns in one table (called a referencing or child or dependent table) that refers to a column or set of columns of another table/tables (called a referenced or parent or base table).
  - The columns from the referencing table must be the primary key in the referenced table. The values of the referencing columns must occur in the referenced table.

- A table can have one or group of columns as a foreign key, and each foreign key can have a different referenced table.
- The referencing and referenced table may belong to the same table, which means the foreign key refers to the primary key of the same table. Such a foreign key is known as self-referencing or recursive foreign key.

Refer to *Figure 3.4* that illustrates the example of referential integrity constraints:

Sid	Fname	Lname	DOB	Salary	……	Did
						1
						1
						3

STAFF (Referencing or Child table)



Did	Name
1	Computer Engineering
2	Information Technology
3	Electronics Engineering
4	Mechanical Engineering

DEPARTMENT (Referenced or parent table)

Figure 3.4: *Example of Referential Integrity Constraints*

- Since DBMS enforces the referential constraints, it must ensure data integrity. For example, if the rows from the parent table are to be updated or deleted, and if the child table/tables still hold, the dependent rows may violate the rules.
- To avoid this dangling effect, different referential actions can be followed to achieve data integrity of the database. They are – CASCADE, RESTRICT, NO ACTION, SET NULL, SET DEFAULT (will be discussed in *Chapter 4*, *Structured Query Language and Indexing*).

- Let us understand the referential integrity constraints with the help of an example. In Figure 3.4, the primary key attribute of table DEPARTMENT is Did. In order to link the record with the STAFF tables, Did is being used as a foreign key in the STAFF table.
- Referential integrity is achieved through the enforcement of foreign key constraints. The outcome of referential integrity is no dangling references.
- It means it will allow only the existing tuples from the Department relation set to participate into the Works_in relationship.
- If a particular department needs to be deleted, then in the staff table, the referencing tuples must be updated (either by setting NULL or by assigning another existing Did from the department table).

B. Implicit constraints or inherent model-based constraints

Constraints that are inherent in the data model and need to be enforced are referred to as inherent model-based constraints or implicit constraints.

The constraint that a relation cannot have duplicate tuples is an inherent constraint. For example, Tuple Uniqueness Constraint, which implies that within any relation all tuples must be unique. Consider the COURSE relation, as shown in *Table 3.2(a)*:

Cid	**Cname**	**Sem**	**Credits**
ITC301	Database Management System	III	4
ITC202	Operating System	IV	4
ITC705	Internet of Things	VII	3

Table 3.2(a): *Course relation*

In the preceding relation, all the tuples are unique because Cid is the key attribute and uniquely identifies the tuples from the relation; hence, it satisfies the tuple uniqueness constraint. However, in the Course relation, as shown in Table 3.2(b), the redundant entry of Cid would violate the tuple uniqueness constraint:

Cid	Cname	Sem	Credits
ITC301	Database Management System	III	4
ITC202	Operating System	IV	4 -------------- not unique
ITC202	Operating System	IV	4 -------------- not unique
ITC705	Internet of Things	VII	3

Table 3.2(b): *Course relation*

C. Semantic constraints or business rules

- Constraints that cannot be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs are referred to as application-based or semantic constraints or business rules.
- The first section of this chapter shows the inherent constraints on the relation. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint. The constraints

that can be expressed in the schema of the relational model via DDL are explicit constraints. The business rule constraints are more general in nature. It is difficult to model or enforce them within the data model. So, they are usually checked within the application programs during the database up-dations.

- The functional data dependency and *multivalued dependency* is another important constraint on the relational database. The functional data dependency is related to the normalization, and we will cover that in detail in *Chapter 5*, *Relational Database Design*. The process of normalization is the goodness of fit test for the relational databases.

Database schema

Database is a collection of relations. A relation can be an entity, a multivalued attribute or a relationship.

When we talk about a database, we must differentiate between the database schema, the logical design of the database, and a database instance.

The concept of a relation schema corresponds to the programming language notion of type definition.

A variable of a given type has a particular value at a given instant in time. Thus, a variable in the programming languages corresponds to the concept of an instance of a relation.

It is convenient to give a name to a relation schema, just as we give names to the type definitions in the programming languages. We adopt the convention of using uppercase names for relations and the attribute names beginning with an uppercase letter for relation schemas.

Following this notation, we use EnggCollegeDatabase to denote the relation schema for Engineering College. Thus the STUDENT relation with attributes can be represented as:

STUDENT

Rollno	Fname	Lname	City	State	Pin	Email_id	DOB	Did

- In general, a relation schema is a list of attributes and their corresponding domains.
- To define the relation schema for the relation STUDENT, we must specify the domain of each attribute.

Concept of keys

A database is a collection of interrelated relations. A relation in the database gives the information about the entity, attribute, or a relationship. Key is a set of one or more attributes whose combined values are unique among all the occurrences of the given dataset. A key in a relational database used for ensuring uniqueness. Keys ensure data integrity and are used to establish the relationship among relations/tables. For example, Cid (Course id) is a key attribute, as shown in *Table 3.1*. In this section, we will discuss the various keys of DBMS.

In short, we require a key for a relation to do the following:

- Uniquely identify a row/record/tuple from a table/relation.
- Keys are used to establish a relationship between tables/relations.
- To enforce and maintain data integrity.

- If the key attributes are mentioned properly during the database design process, it will surely help the designer to come up with an efficient and error-free database schema.

Types of keys in relational database model

The following are the types of keys in a relational database model:

A. Super key

- Super key is a key which uniquely identifies the tuples from relation/table.
- It is a set of one or more attributes from a relation which uniquely identifies the tuples from a relation.
- A super key may have additional attributes which are not required for the unique identification of rows.
- For example, Cname, i.e., course name is a unique attribute of the COURSE relation which uniquely identifies a course in a department. But this cannot uniquely identify a course in an engineering college. For example, the Database management system course is the same in two departments (i.e., IT and Computer). So, to uniquely identify tuples, there is a need to combine it with the department names.
- For example, in a relation STAFF, the Name of the two staff members might be the same in a department; however, their Sid (i.e., Staff ID) cannot be the same. Hence, its combination can be used to identify the tuples uniquely.
- So, Super key would be – { Sid} , { Sid, Name} , { Sid, Name, DOJ} , etc.

B. Candidate key

- A candidate key is a key which uniquely identifies the tuples from a relation/table.
- Candidate key is a super key whose values are not repeated in the table records.
- A minimal super key is a candidate key, i.e., a candidate key is a minimal super key with no redundant attributes.
- Candidate key is a minimal super key, i.e., if we remove any attribute from the candidate key, it will not be a key.
- For example, the super key { Sid} , can also be termed as a candidate key because the values for Sid are not redundant in the Sid column of the table.

  The composite super key { Sid, Name} cannot be considered as a candidate key because Sid itself is a candidate key.

C. Primary key

- Primary key is a key which uniquely identifies a tuple from a relation/table.
- A table cannot have more than one primary key.
- Primary key is a randomly selected candidate key by the database designer.
- The following are the characteristics of the primary key:
  - Its value can't be a duplicate which means the same value can't appear more than once in the same column.
  - Its value cannot be NULL.
  - It ensures entity integrity constraints.

- For example, Rollno. of a student may be selected as a primary key for identifying a student uniquely from the department as the students enroll in the department.

D. Alternate key

- Candidate key, which is not selected as the primary key, is called an alternate key.
- Alternate key = Candidate keys - Primary Key
- For example, in the STAFF relation, {Sid}, {Contact_number}, and {Email} are the candidate keys, since they can uniquely identify the tuples. If we choose the Sid candidate key as the primary key of the STAFF relation, then the remaining two candidate keys, {Contact_number} and {Email}, would be the alternate keys.

E. Foreign key

- Primary key of one relation becomes the foreign key of another related relation.
- A foreign key is used to define the required data relationship between the tables. It is the set of fields in one relation that is used to "*refer*" to a tuple in another relation.
- It is used for referential integrity within the databases. Unlike the primary key of any given relation, a foreign key can be NULL or redundant. But redundancy due to the foreign keys is required and under control. Hence, it will not enforce a uniqueness constraint.
- It represents a "*logical pointer*".
- The table containing the foreign key is called the referencing or child table, and the table containing the candidate/primary key is called the referenced or parent table.
- The foreign key value of a tuple must represent an existing tuple in the referred relation.

Relational Algebra

A relational model has the following two formal languages:

- Relational algebra
- Relational calculus

Since relations are sets, all of the usual set operations are applicable to them.

Codd proposed relational algebra as a basis or theoretical way of manipulating the relation/table contents. Hence, relational algebra is the basic set of operations used in the relational model.

These operations can be used as relational algebra expressions by the users to query the database for data retrieval.

The result of the query is a new relation, which may be retrieved from one or more relations. These resultant relations can be further modified using algebraic operations.

A sequence of relational algebra operations devise a relational algebra expression, whose result will be a relation.

Relational algebra consists of various groups of operations, which are as follows:

- Unary relational operations (Select, Project, Rename)
- Set theory operations in relational algebra (Union, Intersection, Set difference, Cartesian product)

- Binary relational operations (Join – several variations of Joins, Division)
- Additional relational operations (Generalized project, Outer join, Aggregate functions)

A query is nothing but a sequence of operations on data in relations.

The relational algebra can be used to represent the query execution plans. If one can understand the internals of query execution, then the optimization of query is possible.

Some operations on relation may result into relation, whereas some may not result into relation. For example, the union of a binary relation and a ternary relation is not a relation.

The operations in relational algebra are listed in *Table 3.3*:

Operation	Notation/Symbol	Description
Select	**σ**	Selects a subset of rows from a relation; may be based on certain conditions.
Project	**π**	Projects to a subset of columns from a relation.
Rename	**ρ**	Renames either the relation name or the attribute names, or both.
Union	∪	Unification of records in participating relations.
Intersection	∩	Common records in participating relations.
Difference	**-**	Set of records from the first relation but not from that of the second relation.
Cartesian Product	**x**	The cross product of two relations.
Division	÷	Entities which are interacting with all the entities. Division operator is used to evaluate the queries which contain the keyword 'ALL'.
Joins	⋈	Combining the columns from one or more relations which satisfy the join condition.

Table 3.3: *Summary of relational algebra operations*

Unary relational algebra operations

A. The SELECT Operation

- Select operation is denoted by the symbol "σ" (Sigma).
- It is a unary operator, i.e., it takes only one relation as the operand.
- The syntax for the `SELECT` operation is as follows:

  $\sigma_{<selection\ condition>} (R)$

  Here, the symbol σ (sigma) denotes the `SELECT` operator, R is the relation of the database, and the selection condition is a Boolean expression (condition) specified on the attributes of relation R.

The Boolean expression specified in <selection condition> can have a number of clauses of the following form:

<attribute name><comparison op><constant value>

or

<attribute name><comparison op><attribute name>

Here, `<attribute name>` is the name of an attribute of R, `<comparison op>` could be one of the operators $\{=, <, <=, >, >= \neq\}$, and `<constant value>` is a static value stored in the attribute.

Clauses can be connected by the standard Boolean operators {AND, OR, and NOT} to combine the selection conditions.

Selection returns a subset of tuples from relation which fulfils the condition/expression. It can be imagined as a horizontal partition of the relation into two sets of tuples—one that satisfies the condition and are selected in a resultant relation, and another that does not satisfy the condition and are not part of the resultant relation.

For example: Based on EnggCollegeDatabase if we fire the relational algebra query:

Query 1: "*To select all the courses whose Credit is 3*", we can specify this condition with the select operation, as follows:

$$\sigma_{credits\ =\ 3}\ (COURSE)$$

Refer to *Table 3.4* for the result of the select operation:

Cid	Cname	Sem	Credits	Did
CC503	Web Technologies	V	3	1
CC304	Open Source Technology Lab	III	3	1
ITC705	Internet of Things	VII	3	2
EC502	Embedded System	V	3	3
MC603	Robotics	VI	3	4

Table 3.4: *Result of SELECT operation*

Table 3.4 shows the output of the Query 1 and returns only those tuples from the COURSE relation where the course credits are equal to 3.

Query 2: "*To select the STAFF tuples who live in* the *Mumbai city*", we can specify this condition with the select operation as follows:

$$\sigma_{\ City='Mumbai'}\ (STAFF)$$

The `Select` operation properties are as follows:

- The output relation (S) would have the same schema as the input relation (R), that is, the degree of the resultant relation is the same as that of the queried relation.
- The `SELECT` operation is commutative, which means even if we change the sequence of the operations, at the end, the output would be always the same (i.e., the order of operations does not matter). Refer to the following example:

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

- Sequences of the `Select` operations can be combined into a single `Select` operation using the logical operator (AND), as shown as follows:

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ldots)) = \sigma_{<cond1>\ AND<cond2>\ AND\ldots AND\ <condn>}(R)$$

- The number of tuples in the resulting relation (S) is always less than or equal to the number of tuples in R, that is, the number of rows in the resultant relation is less than or equal to the original queried relation.

B. The PROJECT Operation

Project operation is denoted by the symbol “π” (pi).

Project is a Unary operator, i.e., it takes only one relation as the operand.

The PROJECT operation also acts as a filter and selects certain columns from a relation that is listed in the relational algebra query.

It can be imagined as a vertical partition of the relation into two sets of columns — one that wants in a resultant relation, and another that does not require and is not part of the resultant relation.

The syntax for the PROJECT operation is as follows:

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

Here, the symbol π (pi) represents the PROJECT operation and <attribute list> is the desired set of attributes from the attributes of the input relation (R).

For example, to list each staff’s first name, DOJ, and salary, we can specify this condition with the project operation, as shown as follows:

$$\pi_{Fname,\ DOJ,\ Salary}(STAFF)$$

Query 3: “*To select the Sid, Fname, City, DOJ, and Email ID from STAFF relation*.”, we can specify this condition using the project operation, as follows:

$$\pi_{Sid,\ Fname,\ City,\ DOJ,\ Email}(STAFF)$$

Refer to *Table 3.5* that shows the result of the project operation:

Sid	Fname	City	DOJ	Email
101	Sana	Mumbai	8-06-2008	san@abc.edu
102	Ammar	Pune	8-06-2012	amm@abc.edu
103	Uzair	Vashi	8-06-2010	uzi@abc.edu
104	Seema	Mumbai	14-01-2010	seem@abc.edu
105	Bilal	Pune	14-01-2010	bil@abc.edu
201	Hariram	Mumbai	14-07-2017	hari@abc.edu
202	Archana	Vashi	2-1-2004	archana@abc.edu

203	Imran	Mumbai	18-12-2018	Imran@abc.edu
204	Himani	Vashi	2-01-2018	Himani@abc.edu
205	Sujit	Nerul	14-7-2018	Sujit@abc.edu
301	Sanjay	Mumbai	2-06-2008	sanjay@abc.edu
302	Binoy	Vashi	1-01-1999	Binoy@abc.edu
303	Ravi	Panvel	5-08-2002	Ravi@abc.edu
401	Simran	Pune	5-08-2002	sim@abc.edu
402	Ravi	Pune	1-12-2010	seem@abc.edu
403	Zarah	Mumbai	1-12-2012	zaru@abc.edu

Table 3.5: *Result of PROJECT operation*

Table 3.5 shows the output of Query 3 and returns only the projected columns from the STAFF relation.

Project operation enforces duplicate elimination. In the relational algebra query, if the attribute list includes only the nonkey attributes of the input relation(R), then the duplicate tuples may occur in the output relation (S). However, the PROJECT operation removes the duplicate tuples, and the output relation contains a set of distinct tuples.

The project operation properties are as follows:

- The number of tuples in an output relation is always less than or equal to the number of tuples in the input relation.
    - If the list of attributes includes a key attribute of the input relation (R), then the number of tuples in the output relation (S) is equal to the number of tuples in R.
- The Project operation is not commutative. This means we cannot change the sequence of operation in the expression, as shown as follows:

$$\pi <list1> (\pi <list2> (R)) =! \pi <list2> (\pi <list1> (R))$$

- The preceding expression is true if and only if <list2> contains the attributes in <list1>; otherwise, the left-hand side will not be valid. This is the reason that the PROJECT operation is not commutative.

Single expression versus sequences of relational operations:

Sometimes, we need to apply a series of relational algebra operations which results in the relational algebra expression. So, we can write either a single relational algebra expression by nesting the operations, or we can apply one operation at a time and store this intermediate result relation and give a name to the intermediate output relation.

Query 4: Extract staff ID, name, and DOJ of all the staff members whose salary is less than 80000.

Method I: (Single expression)

$$\Pi_{Sid,\ Fname,\ DOJ} (\sigma_{salary < 80000} (STAFF))$$

Method II: (Sequence of expression)

$$STAFF\_SAL = \sigma_{salary < 80000} (STAFF) \text{ .... (a)}$$

$$RESULT = \Pi_{Sid, Fname, DOJ} (STAFF\_SAL) \text{ .... (b)}$$

- It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.
- In general, it is preferable not to change the sequence of the operation in a relational algebra expression. From the preceding example, it is observed that we applied the Selection and then the Project operation. However, we cannot change the sequence of the operation in this case.
- The sequence of expression can be achieved, as shown in method II with step (a) and step (b). Table 3.6 shows the result of step (a) and Table 3.7 shows the result of step (b):

Sid	Fname	Lname	City	Pin	Gender	Designation	DOB	DOJ	Salary	Email	Did	Sid_AC
102	Ammar	Ansari	Pune	411002	M	Assistant Professor	2-02-1986	8-06-2012	75000	amm@abc.edu	1	104
204	Himani	Javale	Vashi	400706	F	Assistant Professor	14-02-1995	2-01-2018	56000	Himani@abc.edu	2	202
205	Sujit	Chavan	Nerul	400711	M	Assistant Professor	6-06-1988	14-7-2018	52000	Sujit@abc.edu	2	202
403	Zarah	Shaikh	Mumbai	400008	F	Assistant Professor	06-121985-	1-12-2012	75000	zaru@abc.edu	4	403

Table 3.6: *STAFF_SAL relation [Method II - step(a)]*

Sid	Fname	DOJ	Salary
102	Ammar	8-06-2012	75000
204	Himani	2-01-2018	56000
205	Sujit	14-7-2018	52000
403	Zarah	1-12-2012	75000

Table 3.7: *RESULT relation [Method II - step(b)]*

C. The RENAME Operation

The Rename operation is denoted by the symbol "ρ" (rho). It is a unary operation.

It can be used to rename either the relation name, or the attribute names, or both.

Renaming is useful in breaking a complex relational algebra expression.

The syntax for the RENAME operation is as follows:

$\rho_{S\ (A1,\ A2,\ ...,\ An)}$ (R) .... will rename both the relation and the attributes

or

ρ_S (R) .... will rename only the relation

or

ρ (A_1, A_2, ..., A_n) (R) .... will rename only the attributes

Here, the symbol ρ (rho) represents the RENAME operator, A1, A2,...An are new attribute names of the relation R, and S is the new name for relation R.

For example, in the college database, consider the COURSE (Cid, Cname, Sem, Credits) relation, as follows:

Case 1 - Renaming a relation

$$\rho_{\mathit{SUBJECT}}\ (\mathit{COURSE})$$

Case 2 - Renaming an attributes

$$\rho_{\mathit{(Sub\_id,\ Sname,\ Semester,\ Credit)}}\ (\mathit{COURSE})$$

Case 3 - Renaming both relation and an attributes

ρ SUBJECT (Sub_id, Sname, Semester, Credit) (COURSE)

Relational algebra operations from set theory

- The set theory operations are the **union, intersection**, and **minus** operations.
- The union, intersection, and difference are binary operations, i.e., it requires two operands/relations.
- To perform these operations, the operand must be union compatible.
- Union compatible means relations/tables having the same number of attributes (i.e., arity) with corresponding domains (same data types).

A. Union operation

- Union operation is a binary operation, and it is denoted by '∪'.
- The result of union operation is a relation with the same schema as that of input operand relations/tables.
- Consider that the union operation applies on relation A and B, i.e., A ∪ B.
    - For applying union operations on relations A and B, both relations must be type compatible (or union compatible).

- Type compatible means both relations A and B should have an equal number of attributes and also each corresponding pair of attributes should have the same domain.
- The result of A ∪ B would be a relation which includes all the tuples that are in A or in B or in both. It eliminates duplicate tuples.

- Union operation does not change the arity of the resultant relation.
- Union operation may change the cardinality of the resultant relation.
- For example, to retrieve the Cid of all the courses who are either offered by department ID 1 or all the courses whose credits are 3, we can use the UNION operation, as follows:

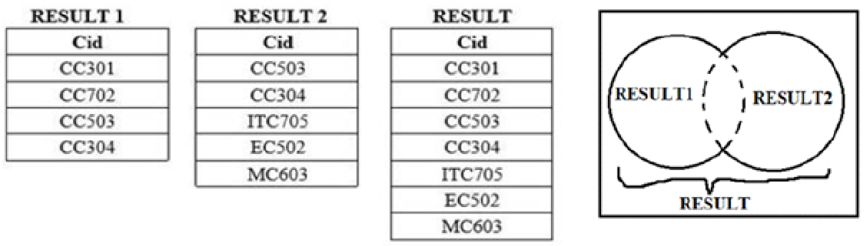
$$RESULT1 \leftarrow \pi_{Cid\ (\sigma\ Did=1)}(COURSE)$$

$$RESULT2 \leftarrow \pi_{Cid\ (\sigma\ Credits=3)}\ (COURSE)$$

$$RESULT \leftarrow RESULT1\ \cup RESULT2$$

- The relation RESULT1 has the Cid of all the courses which are offered in Did 1, whereas RESULT2 has the Cid of all the courses whose Credit is 3. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both, as shown in Figure 3.5. As the union eliminates the duplicates, here Cid values ‘CC503’ and ‘CC304’ appear only once in the result.

  Refer to *Figure 3.5* that illustrates the result of union operation:



RESULT 1

Cid
CC301
CC702
CC503
CC304

RESULT 2

Cid
CC503
CC304
ITC705
EC502
MC603

RESULT

Cid
CC301
CC702
CC503
CC304
ITC705
EC502
MC603

RESULT1
RESULT2
RESULT

Figure 3.5: *Result of Union Operation*

B. Intersection operation

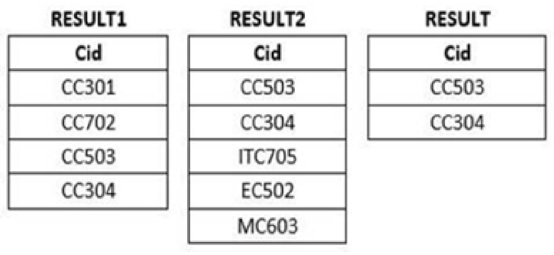
- Intersection operation is a binary operation, and it is denoted by ‘⋂’.
- Consider that the intersection operation applies on relation A and B, i.e., A ⋂ B.
  - For applying the intersection operations on relations A and B, both must be union compatible.
  - The result of A ⋂ B would be a relation which includes all the tuples that appear in both the relations A and B. It eliminates duplicate tuples.
- Intersection operation does not change the arity of the resultant relation.
- Intersection operation may change the cardinality of the resultant relation.
- For example, to retrieve the Cid of all the courses offered by department ID 1 and all the courses whose credits are 3, we can use the INTERSECTION operation, as follows:

$$RESULT1 \leftarrow \pi_{Cid\ (\sigma\ Did=1)}(COURSE)$$

$$RESULT2 \leftarrow \pi_{Cid\ (\sigma\ Credits=3)}\ (COURSE)$$

$$RESULT \leftarrow RESULT1 \cap RESULT2$$

Refer to *Figure 3.6* that illustrates the result of intersection operation:



RESULT1
Cid
CC301
CC702
CC503
CC304

RESULT2
Cid
CC503
CC304
ITC705
EC502
MC603

RESULT
Cid
CC503
CC304

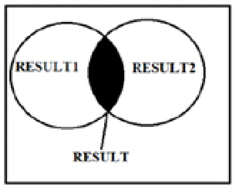


Figure 3.6: *Result of Intersection Operation*

- The relation RESULT1 has the Cid of all the courses which are offered in Did 1, whereas RESULT2 has the Cid of all the courses whose Credit is 3. The INTERSECTION operation produces the tuples that are common in both the relations, RESULT1 and RESULT2, as shown in *Figure 3.6*.

C. The SET DIFFERENCE operation (or MINUS)

- Minus operation is a binary operation, and it is denoted by '-'.
- Consider that the minus operation applies on relation A and B, i.e., A - B.
    - For applying this operation on relations A and B, both must be union compatible.
    - The result of this operation, denoted by A – B, is a relation that includes all tuples that are in A relation but not in B relation.
- The difference operation does not change the arity of the resultant relation.
- The difference operation may change the cardinality of the resultant relation.
- It is not a commutative operation, i.e., A - B ≠ B - A.
- Note that INTERSECTION can be expressed in terms of union and set difference, as shown as follows:

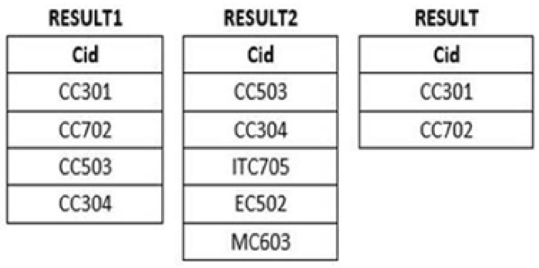
$$A \cap B = ((A \cup B) - (A - B)) - (B - A)$$

- For example, consider the two relations RESULT1 and RESULT2 from *Figure 3.6* in the preceding example, and let's apply the minus operation on the RESULT1 and RESULT2, as follows:

  Case 1: RESULT1 - RESULT2

$$RESULT \leftarrow RESULT1 - RESULT2$$

Refer to *Figure 3.7* that illustrates the output of Case 1:



RESULT1
Cid
CC301
CC702
CC503
CC304

RESULT2
Cid
CC503
CC304
ITC705
EC502
MC603

RESULT
Cid
CC301
CC702

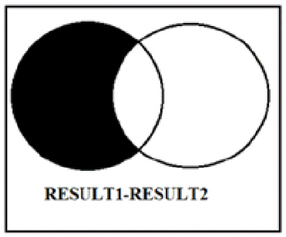
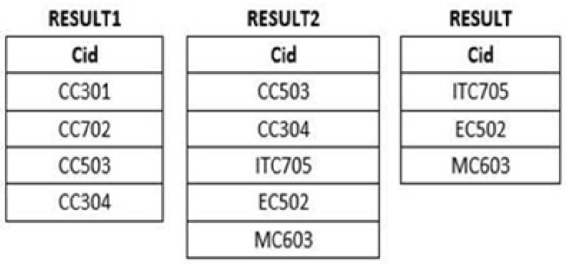


Figure 3.7: *Output of RESULT1 - RESULT2 (Case 1)*

Case 2: RESULT2 - RESULT1

$$RESULT \leftarrow RESULT2 - RESULT1$$

Refer to *Figure 3.8* that illustrates the output of Case 2:



RESULT1

Cid
CC301
CC702
CC503
CC304

RESULT2

Cid
CC503
CC304
ITC705
EC502
MC603

RESULT

Cid
ITC705
EC502
MC603

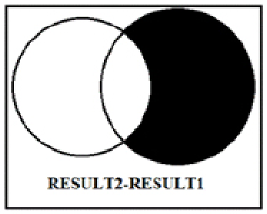


Figure 3.8: *Output of RESULT2 - RESULT1 (Case 2)*

From the preceding examples, according to case 1 and case 2 and the results shown in *Figure 3.7* and *Figure 3.8*, we can observe that RESULT1 - RESULT2 ≠ RESULT2 - RESULT1. Hence, the difference operation is not a commutative operation.

The Cartesian product (cross product) operation

- The Cartesian product operation is also known as cross product or cross join, and it is denoted by 'x'.
- The cross product or cartesian product is a binary operation, i.e., it requires two operand relations.
- Cross product does not require union compatible relations.
- To perform the cross product, each row of the first relation is paired with each row of the second relation.
- It may result in conflict if the attribute/field name is the same in both relations. The relation name may be concatenated with the attribute name to avoid conflict.
- Cross product changes the arity of the resultant relation. The arity is given by the sum of the attributes of the participating relations.
- Cross product changes the cardinality of the resultant relation.
- Consider two relations, A and B, and its cross product would be A x B, as explained as follows:
    - Assume relation A has n attributes, say A_1, A_2, ..., An and relation B has m attributes, say B_1, B_2, ..., Bm. So the resultant relation S would have n + m attributes, say A_1,A_2, ..., A_n, B_1, B_2, ..., B_m, in that order.
    - As a result, relation S will contain every combination of relation A and relation B whether or not they are actually related. Hence, relation S would have n * m tuples.
    - For example, if the first relation has 3 tuples and second relation has 2 tuples, then the resultant has 3 x 2 = 6 tuples.

  Query 5: Extract a list of parent's names and their contact numbers of each student who lives in Mumbai.

  The following is a step-wise implementation of Query 5:

$$STUDENT\_MUM \leftarrow \sigma_{\text{City = 'Mumbai'}} (\text{STUDENT}) \ldots. \text{(c)}$$

$$STUDENT\_LIST \leftarrow \pi_{Rollno,\ Fname} (STUDENT\_MUM) \ \ldots. \ (d)$$

STUDENT_MUM_PARENTS ← STUDENT_LIST x PARENTS …. (f)

Refer to *Table 3.8*, *Table 3.9*, and *Table 3.10* that shows the result of Query 5 for steps (c), (d), and (f):

Rollno	Fname	Lname	City	State	Pin	Email_id	DOB	Did
C101	Sanjay	Jadhav	Mumbai	Maharashtra	400001	sanjay@abc.edu	3-5-1997	1
C102	Sarah	Ansari	Mumbai	Maharashtra	400008	sara@abc.edu	15-6-1998	1
IT201	Siraj	Khan	Mumbai	Maharashtra	400010	siraj@abc.edu	24-5-1997	2
IT202	Sameer	Jadhav	Mumbai	Maharashtra	400008	sam@abc.edu	4-10-1997	2
IT203	Mahira	Ansari	Mumbai	Maharashtra	400008	mahi@abc.edu	27-03-1998	2
E301	Maria	D'souza	Mumbai	Maharashtra	400070	maria@abc.edu	12-12-1998	3

Table 3.8: *Intermediate result in STUDENT_MUM [Query 5 - Step (c)]*

Rollno	**Fname**
C101	Sanjay
C102	Sarah
IT201	Siraj
IT202	Sameer
IT203	Mahira
E301	Maria

Table 3.9: *Intermediate Result in STUDENT_LIST [Query 5 - Step (d)]*

	Rollno	**Fname**	**Rollno**	**Name**	**City**	**Email_id**	**Contact_no**	**Relation**	**Gender**
*	C101	Sanjay	C101	Sameer	Mumbai	sameer@gmail.com	9820445676	Father	M
	C101	Sanjay	C102	Siraj	Mumbai	siraj@gmail.com	9221333444	Father	M
	….	….	….	….	….	….	….	….	….
	….	….	….	….	….	….	….	….	….

	C101	Sanjay	M406	Anushka	Patana	anushka@gmail.com	7070605033	Guardian	F
	C102	Sarah	C101	Sameer	Mumbai	sameer@gmail.com	9820445676	Father	M
*	C102	Sarah	C102	Siraj	Mumbai	siraj@gmail.com	9221333444	Father	M
	….	….	….	….	….	….	….	….	….
	….	….	….	….	….	….	….	….	….

Table 3.10: *Final Result in STUDENT_MUM_PARENTS [Query 5 - Step (f)]*

- In this query, STUDENT_MUM_PARENTS ← STUDENT_LIST x PARENTS.

  The STUDENT_LIST relation has 2 attributes and 6 tuples. However, the PARENTS relation has 7 attributes and 10 tuples. So, the final resultant relation STUDENT_MUM_PARENTS will have 9 attributes and 60 tuples. Out of these 60 tuples, only 6 tuples are actually related (as shown in Table 3.10 through the * symbol) and the other tuples (i.e., 54 tuples) are not related and are unwanted.

- As the cross product also generates unrelated tuples, to make it meaningful, one needs to apply the selection operation after the cross product (which is equivalent to some Join operation).

Division operation

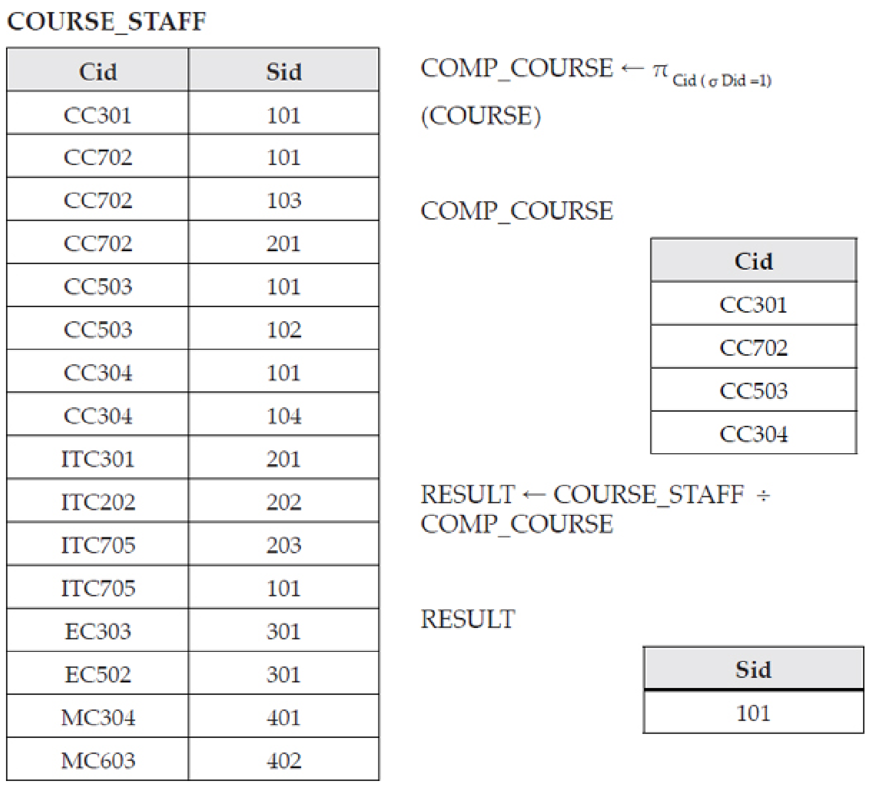
- The Division operation is denoted by '÷'.
- It is a binary operation, however both relations are not required to be union compatible.
- Generally, Division operation can be represented as follows:

  A(X, Y) ÷ B(Y) …….

  It results in X values for those tuples of A (X , Y) for every Y value of relation B.

- For example, retrieve the Sid of the Computer Staff who teaches all the courses in the department, whose Did is 1, i.e., RESULT (Sid) ← COURSE_STAFF (Cid, Sid) ÷ COMP_COURSE (Cid).
- In Figure 3.9, the COMP_COURSE relation contains all the courses (with one column as Cid) offered by the department whose Did is 1, and the COURSE_STAFF relation contains the details about which staff teaches which courses (with two columns as Sid and Cid). Now, we need to retrieve those staff who teach all the courses floated by Did =1.
- The COURSE_STAFF table would be at the numerator and COMP_COURSE table would be at the denominator, and after applying the division operation, it will give Sid of all the staff who teaches in Did =1. So, from Figure 3.9, it can be observed that Sid = 101 is the only faculty who teaches all the courses (CC301, CC702, CC503, CC304) in Did =1. So, Division is a short-hand operation and is used 'for all' types of queries.

  Refer to *Figure 3.9* that shows the output of the division operation:



COURSE_STAFF

Cid	Sid
CC301	101
CC702	101
CC702	103
CC702	201
CC503	101
CC503	102
CC304	101
CC304	104
ITC301	201
ITC202	202
ITC705	203
ITC705	101
EC303	301
EC502	301
MC304	401
MC603	402

COMP_COURSE $\leftarrow \pi_{Cid\ (\sigma\ Did = 1)}$ (COURSE)

COMP_COURSE

Cid
CC301
CC702
CC503
CC304

RESULT $\leftarrow$ COURSE_STAFF $\div$ COMP_COURSE

RESULT

Sid
101

Figure 3.9: *Output of Division Operation*

Join operation

- The Join operation is denoted by '⋈'.
- Join is a relational algebra operation and a derivative of the Cartesian product.
- The Join operation can be considered as a cross product operation followed by a SELECT operation.
- This operation is very useful for any relational database when there is a need to process the relationships among the relations, that is, it is used to join two relations like a Cartesian product, but it removes the duplicate attributes and makes the selection and projection operations simple.
- In other words, it connects the different relations using the columns having comparable information.
- The main difference between the Joins and Cartesian product is that, Join combines the tuples only if it is satisfying the join condition, whereas in the cross product produces all the combinations of tuples.
- The general form of a Join operation on two relations $A(A_1, A_2, \ldots, A_n)$ and $B(B_1, B_2, \ldots, B_m)$ is as follows:

$$A_{<join\ condition>}\ B$$

- The result of the Join is a relation C with n + m attributes $C(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ in that order; C has all the combinations, such as one tuple from A and one from B if it satisfies the join condition.

- Cartesian products may generate quite large results based on the cardinality of the participating relations. In general, it has ($|R| = n, |S| = m \Rightarrow |R \times S| = n * m$) cardinality. Since the combination of the Cartesian product and the selection in queries is common, a special operator join has been introduced.

Note: Relational algebra treats relations as sets, so the duplicate tuples will never occur, neither in input nor in the output of relational algebra operations. But in SQL, the relations are multisets and may contain the duplicates. So, in SQL (will be discussed in *Chapter 4*, *Structured Query Language and Indexing*) it is an explicit operation (SELECT DISTINCT).

Types of Joins

Joins are majorly divided into the following two categories:

- Inner Joins:
  - Natural Join
  - EQUI Join
  - Theta Join
- Outer Joins:
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join

Inner Join

Inner Join includes only those tuples that satisfy the matching condition, while the remaining tuples are excluded. The following are the various types of Inner Joins; let us understand each type with an example:

A. Natural Join

- The natural join is a binary operation. In natural join, we do not need to specify any comparison operator explicitly in condition.
- It combines two different relations based on a common column.
- Most importantly, the common attributes must have the same name and domain in both the relations.
- Natural join acts on those matching attributes where the values of the attributes in both the relations are the same and are put in the resultant relations. It will return the similar attributes only once as their value will be the same in the resulting relation.

Query 6: Display all the courses which have the prerequisite courses.

To implement this query, there is a need to apply the Natural join on the COURSE and COURSE_PREREQUISITE relations. Table 3.11 shows the COURSE relation and Table 3.12 shows the COURSE_PREREQUISITE relation from the *engineering college database* as a ready reference:

Cid	Cname	Sem	Credits	Did
CC301	Database Management System	III	4	1

CC702	Big Data Analytics	VII	4	1
CC503	Web Technologies	V	3	1
CC304	Open Source Technology Lab	III	3	1
ITC301	Database Management System	III	4	2
ITC202	Operating System	IV	4	2
ITC705	Internet of Things	VII	3	2
EC303	Linear Integrated Circuit	III	4	3
EC502	Embedded System	V	3	3
MC304	Machine Design	III	4	4
MC603	Robotics	VI	3	4

Table 3.11: *COURSE relation*

Cid	Prerequisite_Cid
CC702	CC301
EC502	EC302
MC603	MC304

Table 3.12: *COURSE_PREREQUISITE*

Natural Join of COURSE ⋈ COURSE_PREREQUISITE, and its result can be observed in *Table 3.13*.

Cid	Cname	Sem	Credits	Did	Prerequisite_Cid
CC702	Big Data Analytics	VII	4	1	CC301
EC502	Embedded System	V	3	3	EC302
MC603	Robotics	VI	3	4	MC304

Table 3.13: *COURSE ⋈ COURSE_PREREQUISITE*

Table 3.13 shows the Natural join of the two relations COURSE and COURSE_PREREQUISITE. As we can observe, Natural Join combines these two relations based on the common attribute Cid between them. Hence, it appears only once in the resulting relation.

B. Equi Join

Equi join is a special type of natural join of two relations R and S, over a specified attribute. The attributes must be with the same domain. It holds only the equality conditions between a pair of attributes. The resultant relation carries those tuples whose values of the two attributes will be equal and only one attribute will appear in the result.

For example, consider the two relations CAR and BIKE as shown in *Table 3.14* and *Table 3.15*, and their Equi-join result is shown in *Table 3.16*:

Cmodel	Cmake	Cprice
Maruti-800	Maruti	2,80,000
Tata-Nexon	Tata	12, 00,000
Hyundai-Creta	Hyundai	21, 00,000
Maruti-Breza	Maruti	10,00,000

Table 3.14: *CAR Relation*

Bmodel	Bmake	Bprice
Royal Enfield Classic	Royal Enfield	1, 60, 000
Royal Enfield Continental GT 650	Royal Enfield	2, 80, 000
Kawasaki Ninja ZX-10R	Kawasaki	13, 99, 000
Harley-Davidson CVO	Harley-Davidson	51, 35, 000

Table 3.15: *BIKE Relation*

Cmodel	Cmake	Cprice	Bmodel	Bmake	Bprice
Maruti-800	Maruti	2,80,000	Royal Enfield Continental GT 650	Royal Enfield	2, 80, 000

Table 3.16: *CAR ⋈ Cprice = Bprice BIKE*

C. Theta Join

The most general form of inner join is Theta join. The Theta join is denoted as '$\bowtie_\theta$', where Θ is the formula specifying the join predicate. This is the general join which everyone wants to use as it allows you to specify the condition while performing the join operation. Theta join produces the tuples from both the relation operands that satisfy the join condition. But the join condition is other than equality join and involves other comparison operators such as { > , >= , <, <=, ≠ }, because if it is an equality, then it is called the equi join.

A join predicate is specified similar to the selection predicate, except that the attributes are involved from both the participating relations.

For example, the theta join can be specified as R.A '$\bowtie_\theta$' S.B, where A and B are the attributes of the relations R and S respectively. The join of the two relations is equivalent to performing a selection, using the join predicate as the selection formula, over the Cartesian product of the two operand relations. Thus, refer to the following equivalence:

$$R \text{ '}\bowtie_\theta\text{' } S = \boldsymbol{\sigma}_\Theta (R \times S)$$

In the preceding equivalence, we should note that if Θ involves attributes of the two relations that are common to both of them, a projection is necessary to make sure that those attributes do not appear twice in the result.

For example, let us consider the relations CAR and BIKE, as shown in *Table 3.14* and *Table 3.15*. Suppose the customer wants to purchase the CAR as well as a BIKE but doesn't want to spend more money on the BIKE than that of CAR. In this case, we can use the Theta join to obtain the different possible options for the customers. The result can be referred to in *Table 3.17*:

Cmodel	Cmake	Cprice	Bmodel	Bmake	Bprice
Tata-Nexon	Tata	12, 00,000	Royal Enfield Classic	Royal Enfield	1, 60, 000
Hyundai-Creta	Hyundai	21, 00,000	Royal Enfield Classic	Royal Enfield	1, 60, 000
Hyundai-Creta	Hyundai	21, 00,000	Kawasaki Ninja ZX-10R	Kawasaki	13, 99, 000
Maruti-Breza	Maruti	10,00,000	Royal Enfield Classic	Royal Enfield	1, 60, 000

Table 3.17: *CAR.Cprice '$\bowtie_\theta$' BIKE.Bprice*

Outer Join

- An inner join includes only those tuples that have matching attributes and the rest of the tuples are discarded in the resulting relation. Hence, we may lose the rest of the data in some situations. However, Outer join is useful in those cases where we wish to include the information from one or both relations, even if they do not satisfy the join predicate.
- Therefore, we need to use the outer joins to include all the tuples from the participating relations in the resulting relation. If in case it matches, it should join the tuple from both relations, and if it doesn't match, it still joins with padding Null values for the unmatched values.
- Inner join requires the joined tuples from the two operand relations to satisfy the join predicate. In contrast, the outer join does not have this requirement. It is a selective join of either from the left operand (first operand relation), right operand (second operand relation), or both. Outer joins are of three types – Left outer join, Right outer join, and Full outer join – as shown in *Figure 3.10*:

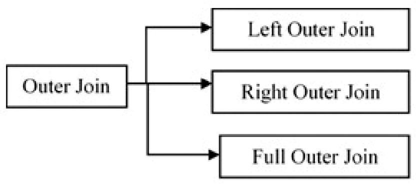


Figure 3.10: *Types of Outer Join*

A. Left Outer Join (R ⟕ S)

All the tuples from the Left relation (say R) are included in the resulting relation. If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are supposed to be set as NULL.

B. Right Outer Join (R ⟖ S)

All the tuples from the Right relation (say S) are included in the resulting relation. If there are tuples in S without any matching tuple in the Left relation R, then the R-attributes of the resulting relation are supposed to be set as NULL.

C. Full Outer Join (R ⟗ S)

All the tuples from both participating relations are included in the resulting relation. If there are no matching tuples for both the relations, their respective unmatched attributes are supposed to be set as NULL.

For example, consider the two relations EMPLOYEE (Ename, Dept, Salary) and EMP_DETAIL (Ename, City, PIN), as shown in *Table 3.18* and *Table 3.19*:

Ename	Dept	Salary
Sonam	Computer	78000
Rahul	Electrical	80000
Zoya	Mechanical	75000
Riya	IT	68000

Table 3.18: *EMPLOYEE relation*

Ename	City	PIN
Sonam	Mumbai	400070
Imran	Kolhapur	401401
Zoya	Sangli	416311
Sai	Pune	410421

Table 3.19: *EMP_DETAIL relation*

The Left Outer Join of EMPLOYEE and EMP_DETAIL (EMPLOYEE ⟕ EMP_DETAIL) can be observed in *Table 3.20*:

Ename	Dept	Salary	City	PIN
Sonam	Computer	78000	Mumbai	400070
Rahul	Electrical	80000	NULL	NULL
Zoya	Mechanical	75000	Sangli	416311
Riya	IT	68000	NULL	NULL

Table 3.20: *Left Outer Join (EMPLOYEE ⟕ EMP_DETAIL)*

The Right Outer Join of EMPLOYEE and EMP_DETAIL (EMPLOYEE ⟖ EMP_DETAIL) can be observed in *Table 3.21*:

Ename	Dept	Salary	City	PIN

Sonam	Computer	78000	Mumbai	400070
Imran	NULL	NULL	Kolhapur	401401
Zoya	Mechanical	75000	Sangli	416311
Sai	NULL	NULL	Pune	410421

Table 3.21: *RightOuter Join (EMPLOYEE ⟖ EMP_DETAIL)*

The Full Outer Join of EMPLOYEE and EMP_DETAIL (EMPLOYEE ⟗ EMP_DETAIL) can be observed in *Table 3.22*:

Ename	Dept	Salary	City	PIN
Sonam	Computer	78000	Mumbai	400070
Rahul	Electrical	80000	NULL	NULL
Zoya	Mechanical	75000	Sangli	416311
Riya	IT	68000	NULL	NULL
Imran	NULL	NULL	Kolhapur	401401
Sai	NULL	NULL	Pune	410421

Table 3.22: *FullOuter Join (EMPLOYEE ⟖ EMP_DETAIL)*

Relational Calculus

Relational calculus is non-procedural or declarative query language. It uses mathematical predicate calculus which is different from that of differential and integral calculus. It is an alternative to relational algebra which is a procedural query language. Relational calculus has no description about how the query will work. It focuses only on what to do and not on how to retrieve it. It allows the user to describe the set of answers without showing procedure about how they should be computed. Relational calculus has a big influence on the design of the commercial query languages such as SQL and Query-by Example (QBE).

Relational calculus is of the following two types:

i. Tuple relational calculus (TRC) proposed by *Codd* in the year 1972
ii. Domain relational calculus (DRC) proposed by *Lacroix* and *Pirotte* in the year 1977

Variables in TRC take tuples (rows) as values and TRC has a strong influence on SQL.

Variables in DRC take fields (attributes) as values and DRC has a strong influence on QBE.

(i) Tuple Relational Calculus (TRC):

Tuple relational calculus was proposed by *Codd* in the year 1972. The calculus is dependent on the use of the tuple variable. The tuple variable ranges over a named relation, that is, in TRC, the set of permitted values of the variables are only tuples from the relation. The tuple relational calculus is a non-procedural query language. TRC gives the desired information without showing the procedure about how it is actually computed. The result is a set of tuples for which a predicate is true. Here, predicate is a truth-valued function with arguments of the predicate calculus.

For example, consider the Staff relation from the *engineering college database*; if we want to list the staff, i.e., tuples from the relation, then we can write Staff(S), where S is a tuple variable of the Staff relation.

Query: To '*find the set of all tuples S such that F(S) is true*,' we can write the following:

$\{ S \mid F(S)\}$

Here, F is called a well-formed formula (wff in mathematical logic). We can add the constraints to filter the tuples from the relation. For example, if the expected result is the list of staff members with salary less than 80K, then we can write the following:

$\{ S \mid Staff(S) \land S.salary < 80000\}$

For example, to list all the staff members from the Department (Did=2) whose salary is less than 80k, we can write the following:

$\{ S \mid Staff(S) \land S.salary < 80000 \text{ AND } S.Did = 2\}$

The tuple variable can be a bound variable or a free variable. When the tuple variable is used with a 'for all' or 'there exists' condition, then it is called a bound variable. In this case, the tuple variable will range over a set of tuples. By changing the tuple variable, the meaning remains unchanged. Any tuple variable without 'for all' or 'there exists' condition is called a free variable.

(ii) Domain relational calculus (DRC):

In the tuple relational calculus, the variable is a tuple in a relation, whereas in domain relational calculus, the variable takes the values from the domains of the attributes.

Domain relational calculus uses the same operators as the tuple calculus. It uses the logical connectives ∧ (and), ∨ (or), and ¬ (not).

It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable.

Notation:

1. $\{ a_1, a_2, a_3, \ldots, a_n \mid F (a_1, a_2, a_3, \ldots ,a_n)\}$

Here, a_1, a_2 are attributes and F stands for formula built by inner attributes.

Example:

For example, suppose from the *engineering college database*, if we select Sid and Fname of the staff who work for Department 1, we can write the following:

```
{<Sid, Fname=""> | <Sid, Fname=""> ? TEACHER Λ Did = 1}
```

For example, suppose from the *engineering college database*, if we get the list of all the courses which have prerequisite courses, we can write the following:

```
∀{ Cname |< Cname> ? COURSE Λ ? Cid( ? COURSE Λ Prerequisite_Cid <> NULL)}
```

Mapping from ER model to relational model

After designing the ER diagram of the system, we need to convert it to relational models which can directly be implemented by any RDBMS like MySQL, Oracle etc.

ER model is a high-level conceptual model of the database design which gives the overall idea about what data will be stored in the database and how they are interrelated.

ER diagrams are mapped to the relational models. To represent the database in the relational schema, standard mapping rules need to be applied.

The rules which are used for mapping an ER diagram into the relational schema, are explained in the following section:

Rule 1 - Strong entity sets and simple attributes

A strong entity within the ER diagram is mapped into a relation. Preferably keep the same name for the relation/table name as an entity set name.

All its attributes can turn into a column (attribute) in the table. The key attribute of the entity set will become the primary key of the table which is usually underlined in the ER diagram.

The following is the process:

- Create a relation/table for each strong entity in the ER diagram.
- Entity's attributes should become the columns of relations/tables.
- Underline attribute (i.e., key attribute) becomes the primary key in each relation/table.

For example, from the *engineering college database*, the COURSE strong entity set can be mapped into a table, as shown in *Figure 3.11;* the COURSE table has 4 columns Cid (Primary key), Sem, Cname and Credits:

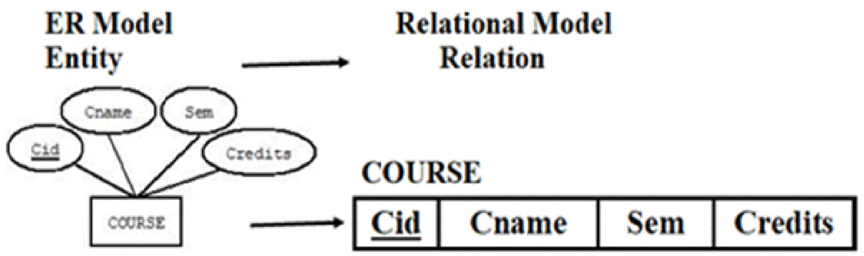


Figure 3.11: *Mapping of a strong entity to relation/ table.*

Rule 2 - Composite attributes

In the relational schema, an attribute of an entity becomes the column name and the intersection of rows and columns, i.e., a cell must have an atomic value.

So, a composite attribute must be further subdivided into parts without changing its sub part meaning. For a composite attribute, create a separate column for each sub part in the given table.

The following is the process:

- Create a separate column for each sub part of the composite attribute.
- Repeat the same process for all composite attributes in the ER diagram.

For example, Name and Address, both are examples of composite attributes. Name has two sub-component attributes, Fname and Lname, whereas Address has two sub-component attributes, City and PIN, in the STAFF table. So, make separate columns for each sub-component. The STAFF ER diagram and its corresponding table is shown in *Figure 3.12*:

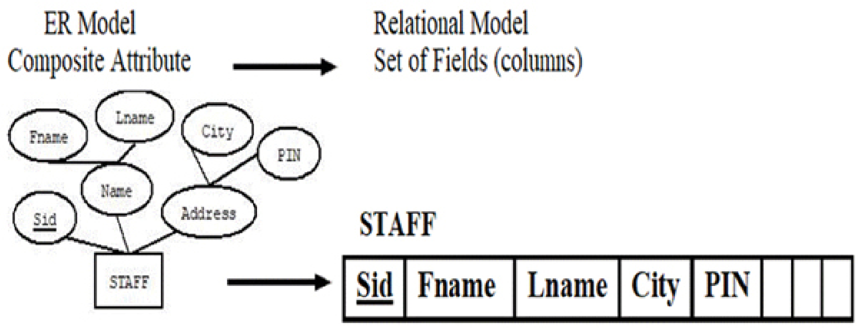


Figure 3.12: *Mapping of Composite Attributes*

Rule 3 - Multivalued attributes

For multivalued attributes, make a separate table with two columns, the first column would be the primary key of the entity set or relationship set, and the second column would be the multivalued attribute. The primary key of a multivalued attribute table is always a composite key (which is the combination of both the columns).

The following is the process:

- Create a separate table with 2 columns for each multivalued attribute, where the first column would be the primary key of the entity set or relationship set, and the second column would be the multivalued attribute.
- The primary key of the resulting table is a composite key, which is a combination of both the columns.
- Repeat the same process for all the multivalued attributes in the ER diagram.

Contact_number is a multivalued attribute of the STAFF entity. So, make a table named STAFF_CONTACT with two columns, first is Sid (i.e., the key attribute of STAFF entity) and second is Contact_number (i.e., the multivalued attribute of STAFF entity), as shown in Figure 3.13. We can give any name to the table; here we gave the name as STAFF_CONTACT. The combination of Sid and Contact_number becomes the composite key of the table. Refer to *Figure 3.13* that illustrates the mapping of multivalued attribute to relational schema:

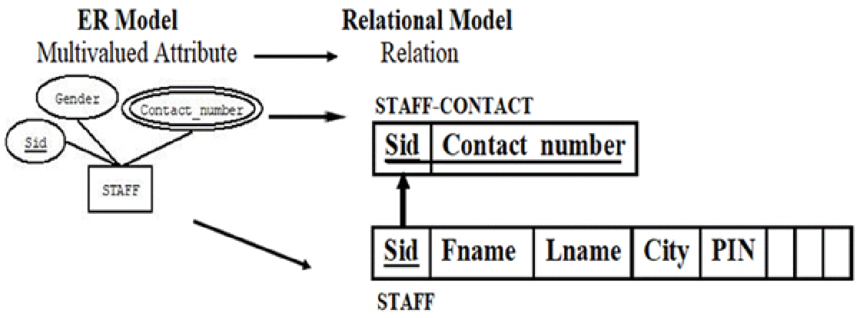


Figure 3.13: *Mapping of Multivalued attribute to relational schema*

Refer to *Table 3.23* that lists the staff contact:

Sid	Contact_number
101	9898989898
101	9797979797
101	9696969696
102	9393939393
103	9222222229
103	9333333339
104	8181818181
...	...
...	...

Table 3.23: *STAFF_CONTACT Relation*

From *Table 3.23*, it can be observed that one staff. whose Sid is 101, has 03 phone numbers for contact. So, it requires to make 03 different entries for Sid = 101 with all its contact numbers. Since, Sid itself is not enough to uniquely identify the tuples, it must combine with Contact_number for the unique identification of the tuples.

Rule 4 - Weak Entity Set

A weak entity set is the one which does not have its own primary key. A weak entity within the ER diagram is mapped into a table. Preferably keep the same name for the table name as the weak entity set name.

Let's assume that A is the weak entity set. This means it does not have its own primary key. B is a strong entity set, on which this weak entity (A) depends, that is, the existence of A entirely depends on the strong entity set B.

While mapping the weak entity from ER to the relational schema, the primary key of the strong entity is used in combination with the unique key of the weak entity for unique identification of the tuples from the relation. The primary key of the strong entities enforces the referential integrity constraints.

The following is the process:

- Create a separate table for the weak entity sets.
- Add all its attributes to the table as columns.
- Add the primary key of the parent strong entity set into the weak entity.
- Enforce the foreign key constraints.
- Repeat the same process for all the weak entity sets in the ER diagram.

For example, in the *engineering college database*, PARENT is the weak entity set which depends on the STUDENT strong entity set. While mapping from ER to the relational schema, the PARENT weak entity set has the PARENT relation, but for the unique identification of the tuples from the weak entity set, it requires a composite primary key which is a combination of the student primary key (Rollno) and the unique key of the weak entity set (Name). Refer to *Figure 3.14* that illustrates the mapping of the weak entity set to the relational schema:

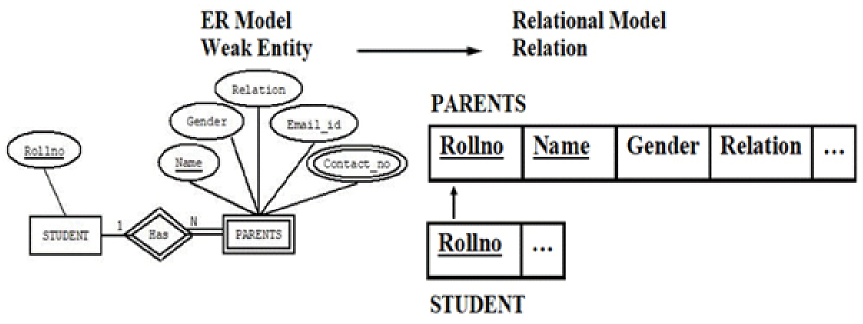


Figure 3.14: *Mapping of Weak Entity Set to relational schema*

Rule 5 - Relationship Set

A relationship is an association among the entities which is used for mapping a relationship set that depends on the participating entity sets and mapping cardinalities.

Based on the participating entity sets and mapping cardinalities, there are the following relationship types:

- Recursive relationship sets
- Binary relationship sets with attributes
- Binary relationship sets (1:1, 1:N and N:M)

Rule 5.1 - Recursive relationship sets

Recursive relationship sets in ER are maps to the relational schema with the referential integrity constraint within the relation or with separate relation.

For example, in the *engineering college database*, we have two recursive relationship sets. First one is Monitor_task relationship which is associated with STAFF relation and second one is Prerequisite relationship associated with COURSE relation.

Rule 5.2 - Binary relationship sets with attributes

The following is the process:

- Create a separate table for the relationship sets.
- If the relationship set has any attribute, add each attribute as a column of the table.
- Add all the participating entity sets' primary key as the column of the table.
- Enforce the referential integrity constraints.
- Repeat the same process for all the relationship sets in the ER diagram.

Case I - 1:1 Relationship type

Refer to *Figure 3.15* that illustrates the ER diagram of 1:1 relationship type:

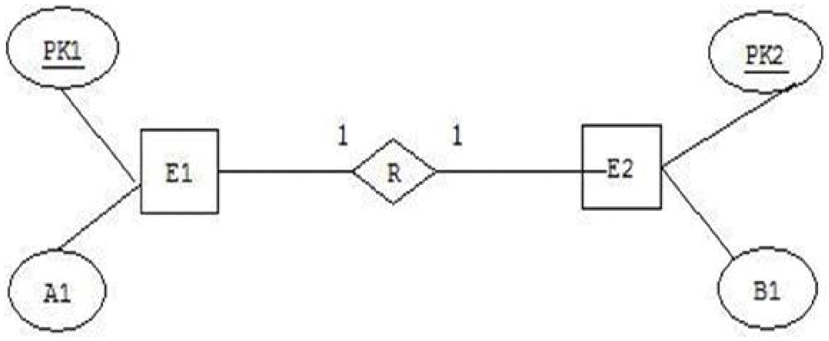


Figure 3.15: *ER diagram of 1:1 relationship type*

If the relationship type 1:1 exists between two entity sets, as shown in *Figure 3.15*, then for mapping into the tables, there are a total four possibilities.

The following are the four possibilities:

Solution 1:

Table Name: E1

PK1	A1

Table Name: E2

PK2	B1	PK1 (as FK)

Solution 2:

Table Name: E1

PK1	A1	PK2 (as FK)

Table Name: E2

PK2	B1

Solution 3: This is one of the possible solutions for 1:1 mapping but is the least preferable option. It is advised and highly recommended to avoid this solution while designing the database schema, as shown as follows:

Table Name: E1

PK1	A1

Table Name: R

PK1 (as FK)	PK2 (as FK)

Table Name: E2

PK2	B1

Solution 4:

PK1	A1	PK2	B1

For example, in the *engineering college database*, one department is controlled by one staff at a time, which is the 1:1 relationship type. While mapping 1:1 relationship type to relational schema, we have 04 possible options, as discussed earlier.

The control relationship set between STAFF and DEPARTMENT, Solution 4 is the most suitable option, as shown in [Table 3.24](#):

Did	**Dname**	**Head_id**	**Start_date**

Table 3.24: *DEPARTMENT relation*

Case II - 1:N Relationship type

Refer to [*Figure 3.16*](#) that illustrates the ER diagram of 1:N relationship type:

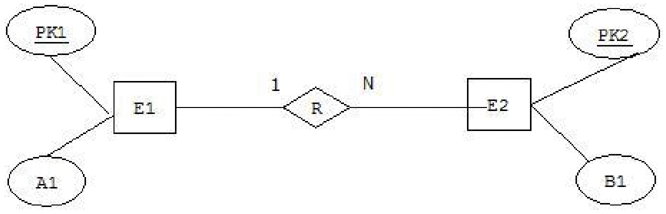


Figure 3.16: *ER diagram of 1:N relationship type*

If the relationship type 1:N exists between two entity sets, as shown in [*Figure 3.16*](#), then for mapping into the tables, there are a total of 2 possibilities, which are shown as follows:

Solution 1:

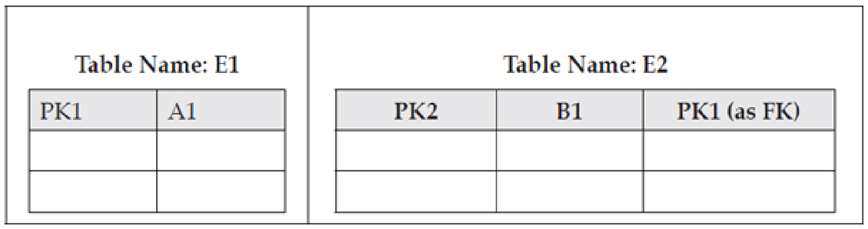


Table Name: E1

PK1	A1

Table Name: E2

PK2	B1	PK1 (as FK)

Solution 2: This is one of the possible solutions for 1:N mapping but is the least preferable option. It is advised and highly recommended to avoid the following solution while designing the database schema:

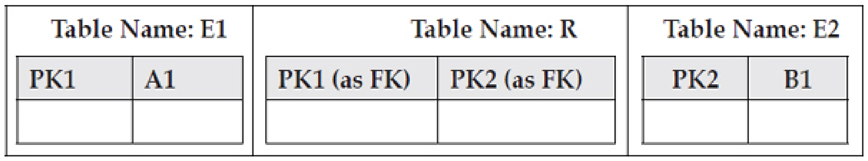


Table Name: E1

PK1	A1

Table Name: R

PK1 (as FK)	PK2 (as FK)

Table Name: E2

PK2	B1

Case III - N:M Relationship type

Refer to [*Figure 3.17*](#) that illustrates the ER diagram of the N:M relationship type:

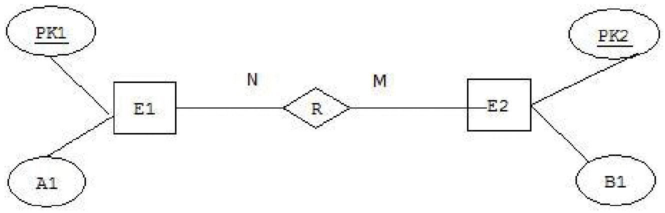
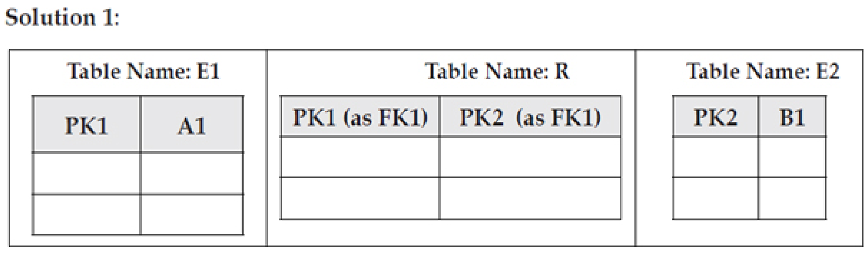


Figure 3.17: *ER diagram of N:M relationship type*

If the relationship type N:M exists between two entity sets, as shown in *Figure 3.17*, then for mapping into the tables, there is only one solution to convert into a table, which is as follows:



Solution 1:

Table Name: E1

PK1	A1

Table Name: R

PK1 (as FK1)	PK2 (as FK1)

Table Name: E2

PK2	B1

Rule 6 -Hierarchical entities/Specialization/Generalization

ER specialization or generalization is represented in the form of the hierarchical entity sets.

The following is the process:

- Create separate tables for the super classes and its attributes become the columns of the table. The key attributes become the primary key of the table.
- Create separate tables for each sub class and its attributes become the columns of the table. The primary key of the superclass should also be added as one more column in the subclass table and enforce the foreign key constraint on it.

For example, with reference to *Figure 2.29* in *Chapter 2*, *The Entity-Relationship Model* Generalization and Specialization can be modelled and mapped into the equivalent relational schema by applying the preceding process, and the resultant relational schema is shown in *Figure 3.18*.

As shown, for Developer and Intern relation, the primary key is the composite key, which is the combination of superclass and subclass primary key.

Refer to *Figure 3.18* that illustrates the mapping of Generalization and Specialization:

Student

Reg	Name	DOB	Address

Developer

Reg	Dev-id	Project	Salary	DOJ

Intern

Reg	Intern-id	Company	Stipend	Duration

Table 3.24: *Mapping of Specialization/Generalization*

Rule 7 - Aggregation

In the ER diagram, a relationship is an association between the entities. In the chain's ER diagram, there is no scope to represent the relationships between the relationships. To overcome this limitation, an EER model concept, aggregation can be used.

Aggregation is an abstraction through which we can represent the relationships as a higher-level entity, as shown in *Figure 3.19*:

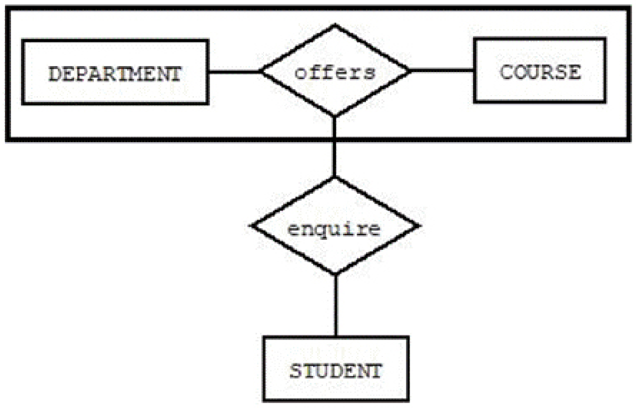


Figure 3.18: *EER diagram - Aggregation*

For example, a department offers different courses for a particular program. Practically, any student will never enquire either about only the course or only the program. Hence, while seeking admission, the students enquire about both, the different programs and the courses offered by the programs, that is, the students are interested to enquire about the current courses offered by the different programs in which s/he is interested. Accordingly, s/he will choose the program to be enrolled.

From *Figure 3.19*, the 'offer' relationship between Department and Course is treated as a high-level entity, which is in the enquire relationship with the Student entity. Aggregation can be used to simplify the ternary relationship in the ER diagram.

The following is the process:

For all entity set, make a separate table, as discussed in Rule 1.

For all the relationship sets, make a separate table (if needed) as discussed in Rule 5.

Create one more table for the relationship set which connects a high-level entity and a strong entity set with all its descriptive attributes and primary key of participating relationship sets.

From the preceding process, i.e., *Figure 3.19*, Aggregation is mapped into 03 tables (DEPARTMENT, COURSE, STUDENT), for each entity set with their attributes and primary key, whereas the high-level entity set is connected with a strong entity set. We need to create one more table ENQUIRE which contains all the primary keys of the participating relationship sets, as shown in *Figure 3.20*:

DEPARTMENT

Did	Dname	Head_id	Start_date

COURSE

Cid	Cname	Sem	Credits	Did

STUDENT

Rollno	Fname	Lname	City	State	Pin	Email_id	DOB	Did

ENQUIRE

Did	Cid	Sid

Table 3.25: *Mapping of Aggregation to Relational Schema*

Conclusion

A relation schema is a list of attributes that describe its structure, and a set of tuples specify the relation state.

The relational model is the widely accepted model for data storage. Here, data is stored in the form of tables.

The informal name of relation is a table, and they are interrelated by referential integrity constraints.

Entity set is a grouping of related entities, and a relationship set is a grouping of related relationships.

In the table, a row represents the tuple/record/entity, column represents the attribute and the intersection of row and column store atomic values.

The central concept of the relational model is the different types of keys. The different keys used in the relational models are primary key, candidate key, super key, unique key, and foreign keys.

There are two types of Integrity constraints – entity integrity and referential integrity. Entity integrity is maintained using a primary key.

The constraints on the referential integrity are that, the foreign key must contain the values that match the primary key in the base table or must contain nulls.

Relational algebra becomes important for the relational models since it is a mathematical notion of a "*relation*". It is used to form the queries and is fundamental to query optimization.

The relational algebra operations are broadly classified as unary and binary operations.

Unary relational algebra operations are Selection, Projection, and Rename. Binary relational algebra operations are Sets theory operations (union, intersection, minus), division, cartesian product, and joins.

Relational calculus is a non-procedural language and has two types – tuple relational calculus and domain relational calculus.

Refer to *Table 3.25* for the mapping of ER, EER to relational schema:

ER/ EER Model	Relational Model
Entity (Strong/Weak)	Relation
Attribute	Field/ Column
Relationship (1:1, 1: N)	Foreign key
Relationship (N:M)	Relation
Specialization/ Generalization	N Entity relations with Primary key in child relation
Aggregation	N Entity relations with relation for relationship

Table 3.25: *Mapping of ER, EER to relational schema*

The relational database became successful because of Structured Query Language (SQL), and it is the primary interface for querying relational databases.

The next chapter describes the SQL in-depth for creating, querying, and maintaining the database.

Questions

1. Which characteristics of the relations make them different from the traditional tables and files?
2. What is a relation? Explain with suitable examples. What are the properties of relation?
3. What is the informal name of relation in the relational model? What role does it play in the relational model?
4. What is the basic structure of the relational models?
5. Define the following terms and explain with suitable examples:
   a) Database schema
   b) Domain
   c) Attribute
   d) Tuple
   e) Domain constraint
6. What is key? Explain the various types of keys in the relational data models.
7. Explain the following terms with suitable examples:
   a) Primary key

b) Candidate key

c) Foreign key

d) Super key

8. What is this concept of foreign key and how can it be applied?
9. What is NULL in a relational database? Why is it required?
10. Discuss the role of NULL in the relational databases.
11. Explain the types of integrity constraints in detail.
12. What are the different integrity rules available in the relational models? Explain with a suitable example.
13. Can we design the relational database without the entity integrity and referential integrity constraints? Why is each one important?
14. What are the rules to convert the ER and EER diagrams into a Relational model?
15. Draw an ER diagram for the Banking system and convert it into a relational model.
16. Draw an ER diagram for the Hospital management system considering doctor, patient, hospital, supporting_staff and services as the major entities (assume the attributes and relationships among them) and convert it into a relational model.
17. List and explain the different mapping rules from ER and EER to relational schema.
18. What is a relationship? List the different relationship types and their mapping to the relational model.
19. Explain the relational algebra operations from Set Theory in detail.
20. Explain the Unary relational operations in detail.
21. Explain the Binary relational algebra operations in detail.
22. Explain the following relational algebra operations with suitable examples:

    a) Division

    b) Cartesian product
23. What is a Join? Explain its types with suitable examples.
24. Write a short note on the following:

    a) Schema diagram

    b) Relational algebra

    c) Relational calculus
25. Consider the following schema:
    SUPPLIERS (<u>Sid</u>, Sname, Address)
    PARTS (<u>Pid</u>, Pname, Color)
    CATALOG (<u>Sid</u>, <u>Pid</u>, Cost)

    Write the following query in relational algebra:

    a) Find Sids and Sname of the suppliers who supply some red and some green parts.

    b) Find Pid and Pnames of the parts that are supplied by at least two suppliers.

    c) Find Snames of the suppliers who supply every part supplied by the ACME corporation.

    d) Find the suppliers who supply the maximum types of parts.

CHAPTER 4

Structured Query Language and Indexing

Donald D. Chamberlin
(Born-Dec 21, 1944) The Father of SQL

Relational databases came into existence with *E.F. Codd's* 1970 landmark publication, "*A Relational Model of Data for Large Shared Data Banks*". Structured Query Language (SQL) is the most widely used commercial relational database language, developed in the mid-1970s by *Chamberlin* and *Raymond F. Boyce*. The duo published the "*SEQUEL: A Structured English Query Language*" paper. SEQUEL is a sublanguage used for data definition and data manipulation facility. The simple block structured syntax and simple operations on the tables enable the user's easy interaction with SEQUEL. Its simplicity is enhanced by the fact that much of its work takes place behind the scenes.

Edgar F. Codd introduced the relational model in a series of papers with simplicity, data independence, and many other advantages. The relational model suggests that the data in the database management system can be represented in the form of tables. A number of languages have been proposed for expressing the queries against a relational database, including the relational algebra and relational calculus proposed by *Codd*. IBM developed the Structured Query Language (SQL), pronounced as either S-Q-L/Sequel and composed of commands to create, manipulate, and administrate the data to extract useful information, as part of the System R project in the early 1970s. All relational DBMS software supports SQL, and many software vendors have developed extensions to the basic SQL command set, for example, MySQL, Oracle, MS Access, and SQL Server use SQL as their standard database language.

In short, SQL has clearly established itself as a programming language that we use to talk with and access our database. SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987. It is a standard language for the Relational Database Management System.

Introduction

The relational database became successful because of SQL. IBM developed the original version of SQL, and since its inception, it has become the de-facto standards for the relational database. SQL is the primary interface for querying relational databases. Even though we refer to SQL as Structured Query Language, it is more than just a "*query language*". It defines the structure of the data, modifies the data in the database, and can specify the security constraints. It is a bit different from the conventional computer programming languages and is easy to learn because of its simple vocabulary. It allows the users to access the data in relational databases in an enhanced way, i.e., much of its work takes place behind the scenes. SQL is about the data and results. A simple command creates the complex database or updates a record without knowing the physical data storage format. With these advantages, there are certain limitations of SQL related to data entry as it does not provide inbuilt menus or report forms. In this chapter, we will discuss the basic functionalities of SQL.

Structure

In this chapter, we will cover the following topics:

- Basic commands and functions of SQL
- SQL for DDL, DML, TCL, and DCL

- Simple and complex queries of SQL
- Aggregate functions, GROUP BY, and HAVING clause
- Cursor, triggers, and embedded SQL

Objectives

After studying this chapter, you will be able to use the SQL commands to obtain the desired results, design and manage the database, and execute the queries related to the application. You will learn how to use the aggregate functions to generate summarized results. You will also study about the implementation of cursor, triggers, and JDBC.

Features of SQL

Commercial SQL is more user friendly and allows the users to create the database and table structures. It allows the users to write the query in simple English-like statements. The SQL provides the basic data management operations such as add, modify, delete, operations to transform the data into information, and specify the security constraints. The main features of SQL are shown in *Figure 4.1*:

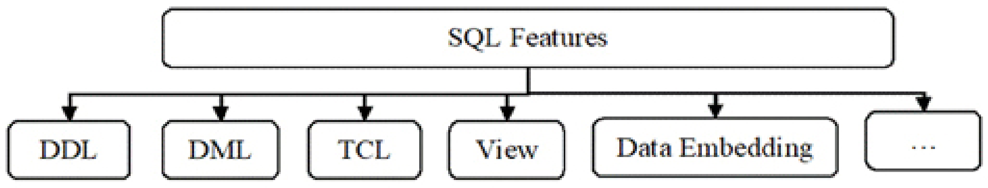


Figure 4.1: *Features of SQL*

The features of SQL, as illustrated in *Figure 4.1*, are defined as follows:

1. **Data Definition Language (DDL)**

   Basically, the DDL commands are used to create and modify the database objects. SQL DDL provides the various commands for defining, modifying, and deleting the existing relation schemas.

2. **Data Manipulation Language (DML)**

   The DML commands are used to manage the data. Data manipulations can be achieved by inserting, deleting, or modifying the tuples.

3. **Data Control Language (DCL)**

   DCL controls the access rights and permissions of the users. The access rights and permissions can be assigned or withdrawn by using the `GRANT` and `REVOKE` commands.

4. **Transaction Control Language (TCL)**

   TCL is used to manage the transactions in the database. The SQL TCL commands are used for specifying the explicit beginning and end of the transactions as well as for providing the commands for Rollback, Savepoint, and Commit.

5. **View definition**

   Views can be considered a virtual table which helps restrict the data access. The SQL DDL includes the commands for defining the views.

6. **Embedded data manipulation language**

   The embedded form of SQL is designed for use within the general-purpose programming languages such as C++, Java, Python, .NET etc., for combining the computing power of a programming language and the database manipulation capabilities of SQL. It helps to relate and integrate the other systems and programs of the real world, such as the Web-based applications.

7. **Data Integrity**

   Data integrity is the assurance of the accuracy and consistency of data in the database. It will protect the data from any unwanted or unintended changes while managing the data.

Advantages of SQL

The following are the advantages of SQL:

Simplicity

- It is easier than the lower-level language as users can write the query in simple English-like statements.
- Optimized query execution will result in increased productivity.

Completeness

- The language is relationally complete, i.e., almost all the related tasks can be achieved using queries and does not require loops or branching.

Non-procedural

- SQL is a nonprocedural language.
- For example, a `SELECT` statement specifies only what data is required and not the procedure to obtain it.
- The user's intent is captured at an abstract level which makes optimization a practical proposition.
- Knowing the user's intent is also important in the system implementation, such as authorization checking.

Data independence

- Data independence is the major aspect of SQL and database. SQL DML provides total "*physical*" data independence.
- In SQL, there are two types of data independence – Physical and Logical.
- The database schema can be modified without affecting its physical storage, i.e., data independence helps the users to keep the data separated from all its programs.

Ease of Extension

SQL provides many built-in functions which can be used to augment the power of the basic SQL language.

Support for higher-level languages

- Database and SQL have different types of users having different requirements. For the Naive users, it is possible to build a tailor-made software which embeds SQL as a part of the higher-level language. It will ease the access of data by enhancing the capability by using the higher-level languages.
- For the programmers, it is easy to embed SQL in the higher-level language of their choice based on the core competencies and requirements.

SQL Data Types

To demonstrate the various features of SQL, we will be using MySQL opensource tool. MySQL supports various data types. It specifies the type of data to be stored in the database.

Based on the data types, it is possible to identify a set of values for a given column. MySQL supports various data types, as shown in *Figure 4.2*:

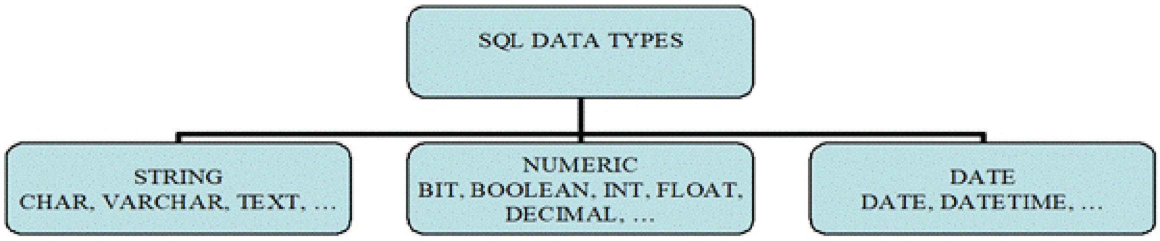


Figure 4.2: *SQL Data Types*

String data types

It contains any value that holds the letters and numbers including the binary data, image or audio files. It basically used to store the textual data such as names, designations, addresses etc. The String Data types allow both fixed and variable length strings.

The following are some common string data types in MySQL:

- **`CHAR(size)`**: It holds maximum 255 characters and allows a fixed length string. Here, the size is the number of characters to store. Specifying the length is not compulsory, and the default size is 1.
- **`VARCHAR(size)`**: It is a variable-length string between 1 and 255 characters in length. Here, the size is the number of characters to store. You must define a length when creating a `VARCHAR` field. Whenever we have a variable length data, Varchar is more appropriate as compared to the Char data type, as it saves memory.
- **`Binary Large Objects (BLOB)`** or **`TEXT`**: It allows a string with a maximum length of 65,535 characters. No need to specify a length with BLOB or TEXT. These data types are used to store large amounts of binary data, such as images or other types of files. Here, BLOB is case sensitive, whereas TEXT is not case sensitive.

Numeric data types

It allows both the signed and the unsigned integers. It is basically used to store data like marks, price, salary etc.

MySQL supports the following numeric data types:

- **`INT:`** It allows signed integers from -2147483638 to 214747483637 and unsigned integers from 0 to 4294967925. We can specify a width of up to 11 digits. It requires 4 bytes for storage.
- **`FLOAT(M,D)`** or **`DOUBLE(M,D):`** `FLOAT` is a single-precision floating point number and `DOUBLE` is a double-precision floating point number. These data types are also known as "*Floating-Point*" data types. Here, (M,D) means that values can be stored with up to M digits in total, of which D digits may be after the decimal point.

Date and Time data types

This data type enables us to mention the date, time, and also the combination of date and time. MySQL supports the following Date and Time data types:

- **`DATE:`** It holds the date values in the format – YYYY-MM-DD – where the supported range is 1000-01-01 to 9999-12-31. It only accepts valid dates between the mentioned supported ranges. It takes 3 bytes for storage.
- **`DATETIME:`** It holds a combination of date and time values in the format – YYYY-MM-DD HH:MI:SS – where the supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. It takes 5 bytes plus fractional seconds for storage.
- **`TIME:`** It stores the time in an HH:MM:SS format, where the supported range is -- 838:59:59 to 838:59:59. It takes 3 bytes plus fractional seconds for storage.
- **`YEAR:`** The `YEAR` data type is used to store a four-digit year value in the YYYY format or two-digit year value in the YY format. The default length is 4. If the length is specified as 2 (for example `YEAR(2)),` `YEAR` can be between 1970 and 2069 (70 and 69). If the length is specified as 4, then `YEAR` can be 1901 to 2155. It takes 1 byte for storage.

NULL Values

SQL supports a special value called the NULL values. It is used to represent the value of the columns which might be currently not known, missing, or not applicable.

Make a note that the NULL value is not equal to zero or space. Basically, a column with a NULL value indicates that it does not have any value.

SQL allows the queries that check whether an attribute value is NULL. Rather than using = or compare an attribute value to NULL, SQL uses IS and IS NOT. This is because SQL considers each NULL value as being distinct from every other NULL value, so the equality comparison is not appropriate.

It is not possible to check for the NULL values with the comparison operators such as =, <, or <>, so SQL uses the IS NULL or IS NOT NULL operators.

Syntax: (for IS NULL)

SELECT column_names

FROM table_name

WHERE column_name IS NULL;

For example, find the `Sid, Fname`, and `Designation` of all the staff who do not have any Academic coordinator:

SELECT Sid, Fname, Designation

FROM STAFF

WHERE Sid_AC IS NULL;

Syntax: (for IS NOT NULL)

SELECT column_names

FROM table_name

WHERE column_name IS NOT NULL;

For example, Display the Name, City and Contact number of those parents whose `Email_id` is available:

SELECT Name, City, Contact_no

FROM STAFF

WHERE Email_id IS NOT NULL;

Data Definition Language

A relational database is a collection of relations (or tables). Here, Entity, Relationship, and Attributes can be represented as relations, and these relations/tables can be created using DDL, i.e., DDL comprises the SQL commands which are used to define the database schema. It is also used to create and modify the structure of the database objects in the database.

The following commands come under DDL:

- `CREATE`
- `ALTER`
- `DROP`
- `TRUNCATE`
- `RENAME`
- `COMMENT`
- `DESC`

CREATE Command

Definition: This command is used to create a database and its objects like database, table, index, views, store procedure, function, and triggers.

Creating a Database: The `CREATE` command is used to create a database in RDBMS.

Syntax:

CREATE DATABASE <Database_Name> ;

Example:

CREATE DATABASE EnggCollegeDB ;

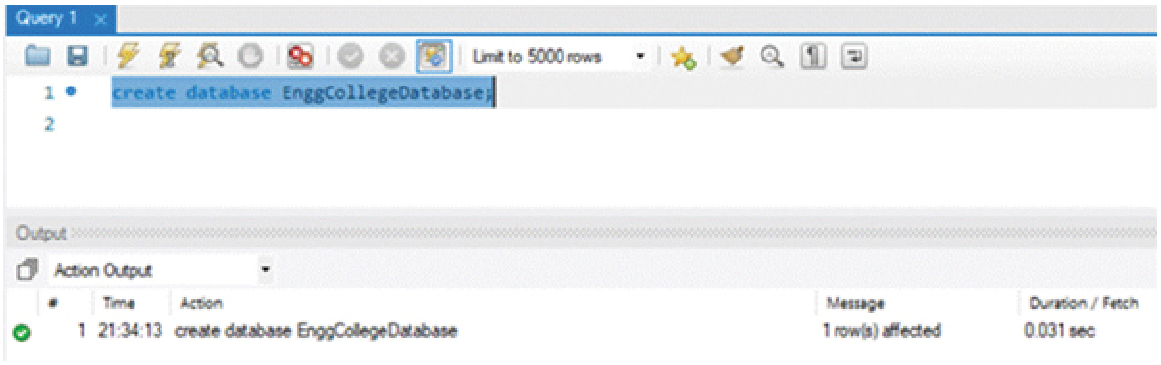
The preceding command will create a database named `EnggCollegeDB` as shown in *Figure 4.3*, which will be an empty schema initially. We can again make use of the `CREATE` command to create tables in this newly created database.

Creating a Table: The `CREATE` command can also be used to create tables. While creating the tables, we always have to specify the details of the columns (such as the name and data type of the columns). The create table command also includes the options to specify certain integrity constraints. A newly created relation is empty initially.

Syntax:

```
CREATE TABLE <TABLE_NAME>
(
    column_name1 datatype1,
    column_name2 datatype2,
    column_name3 datatype3,
    column_name4 datatype4
);
```

Refer to Figure 4.3 that illustrates the `CREATE` database command:



```
create database EnggCollegeDatabase;
```

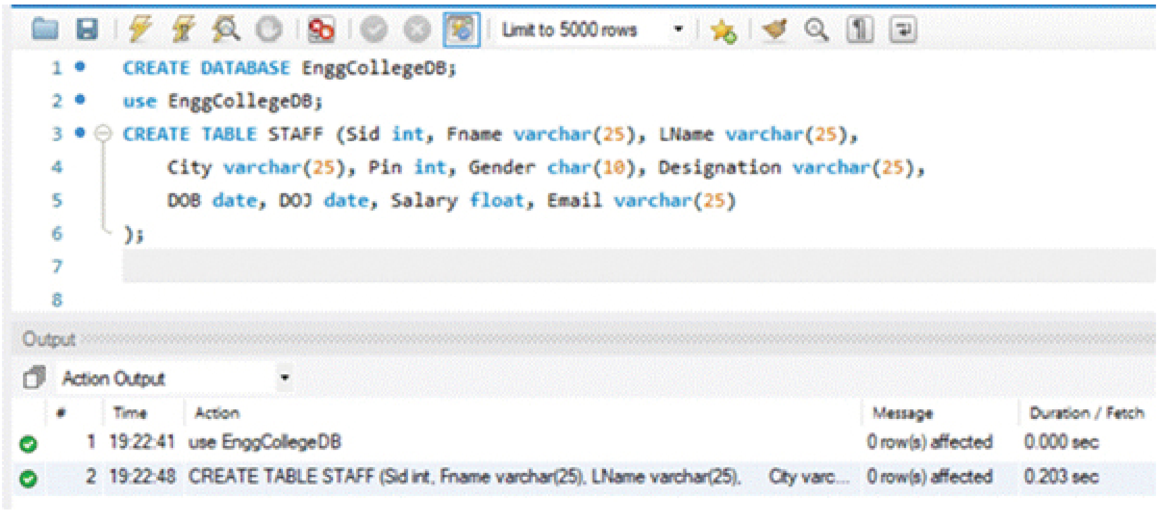
```
#  Time      Action                                Message            Duration / Fetch
1  21:34:13  create database EnggCollegeDatabase  1 row(s) affected  0.031 sec
```

***Figure 4.3:** CREATE DATABASE Command*

Example: Let's create a table called "STAFF" that contains eleven columns – `Sid, Fname, LName, City, Pin, Gender, Designation, DOB, DOJ, Salary and Email` – as follows:

```
CREATE TABLE STAFF
(
    Sid int,
    Fname varchar(25),
    LName varchar(25),
    City varchar(25),
    Pin int,
    Gender char(10),
    Designation varchar(25),
    DOB date,
    DOJ date,
    Salary float,
    Email varchar(25)
);
```

The preceding command will create a table called `STAFF` with eleven columns, as shown in *Figure 4.4*. The Sid and Pin columns are of type int and will hold an integer. The Fname, Lname, City, Designation, and Email columns are of type varchar and will hold the characters and numeric, and the maximum length for these columns is 25 characters. The gender column is of character types which can hold a maximum of ten characters only. The DOB and DOJ columns are of type date which will hold valid dates. Finally, the Salary column is of float type which holds floating-point numbers, as shown in *Figure 4.4*:



```
CREATE DATABASE EnggCollegeDB;
use EnggCollegeDB;
CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),
    City varchar(25), Pin int, Gender char(10), Designation varchar(25),
    DOB date, DOJ date, Salary float, Email varchar(25)
);
```

```
    #  Time      Action                                                            Message            Duration / Fetch
    1  19:22:41  use EnggCollegeDB                                                 0 row(s) affected  0.000 sec
    2  19:22:48  CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),  City varc...  0 row(s) affected  0.203 sec
```

Figure 4.4: *CREATE TABLE Command*

Create a table using an existing table: The `CREATE` command is also used to create a copy of an existing table. The new table gets the same schema definitions as the old table. When a new table is created using an existing table, as can be observed in *Figure 4.5*, the new table will automatically have the existing values from the old table. All columns or specific columns can be selected, which depends on the query.

Syntax:

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE condition ;
```

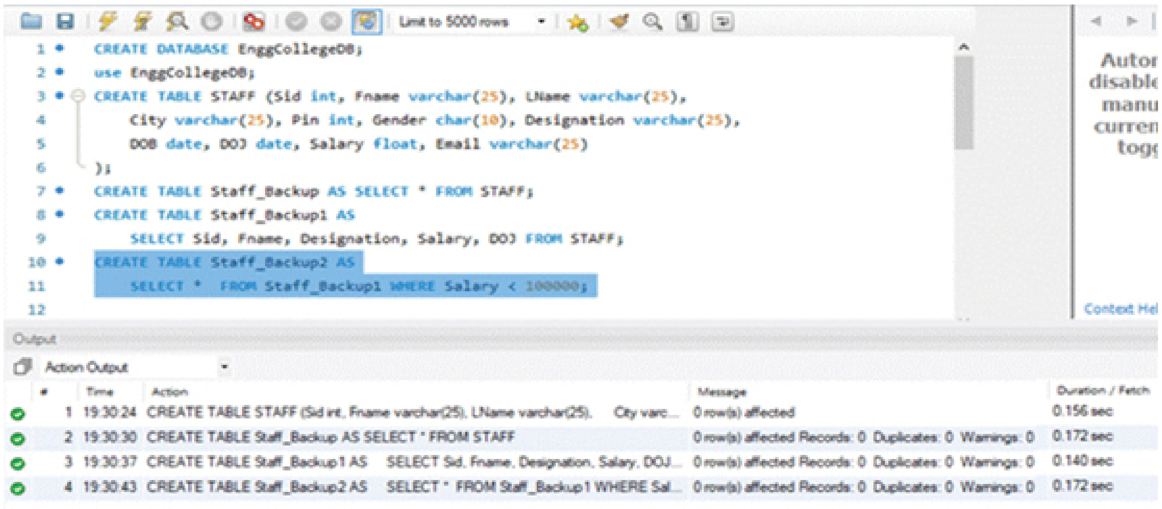
Example: Let's create a new table called "*Staff_Backup*" which will be a copy of the `STAFF` table, as shown in *Table 4.1* and *Figure 4.5*:

CREATE TABLE Staff_Backup AS
SELECT * FROM STAFF;
	It will create an exact replica of the STAFF table.

CREATE TABLE Staff_Backup1 AS
SELECT Sid, Fname, Designation, Salary, DOJ
FROM STAFF;
	It will create a STAFF backup with five attributes from the STAFF table.

CREATE TABLE Staff_Backup2 AS
SELECT * FROM Staff_Backup1
WHERE Salary < 100000;
	It will create a new table Staff_Backup2 with all the columns but Salary is less than 100000 from Staff_Backup1.

Table 4.1: *Create Table using existing Table*



```
CREATE DATABASE EnggCollegeDB;
use EnggCollegeDB;
CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),
    City varchar(25), Pin int, Gender char(10), Designation varchar(25),
    DOB date, DOJ date, Salary float, Email varchar(25)
);
CREATE TABLE Staff_Backup AS SELECT * FROM STAFF;
CREATE TABLE Staff_Backup1 AS
    SELECT Sid, Fname, Designation, Salary, DOJ FROM STAFF;
CREATE TABLE Staff_Backup2 AS
    SELECT *  FROM Staff_Backup1 WHERE Salary < 100000;
```

```
Output
Action Output
 #  Time      Action                                                                 Message                                                  Duration / Fetch
 1  19:30:24  CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),  City varc...  0 row(s) affected                                   0.156 sec
 2  19:30:30  CREATE TABLE Staff_Backup AS SELECT * FROM STAFF                      0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0  0.172 sec
 3  19:30:37  CREATE TABLE Staff_Backup1 AS    SELECT Sid, Fname, Designation, Salary, DOJ...  0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0  0.140 sec
 4  19:30:43  CREATE TABLE Staff_Backup2 AS    SELECT *  FROM Staff_Backup1 WHERE Sal...  0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0  0.172 sec
```

Figure 4.5: *Create Table Using SELECT command (Existing Table)*

DROP Command

Definition: This command is used to delete the objects from the database. For example, database tables, views, index etc. When the `DROP TABLE` command is used, it will drop the table, meaning it will destroy the structure of the table and the data stored in it. The `DROP` command deletes all the information about the dropped relation from the database.

`DROP Table` - It is used to delete both the structure and the records of the table.

Syntax:

DROP TABLE table_name ;

Example: Let's drop the copy of the STAFF table, i.e., drop the `Staff_Backup` table, as follows:

DROP TABLE Staff_Backup ;

The preceding query will delete the `Staff_Backup` table completely, which means delete both the structure and the records stored in the `Staff_Backup` table.

`DROP Database:` It is used to drop the database from the system.

Syntax:

DROP DATABASE database_name ;

Example:

Assume we have a Test database consisting of four tables with some data in each table. Let us destroy the Test database using the `DROP DATABASE` command, as follows:

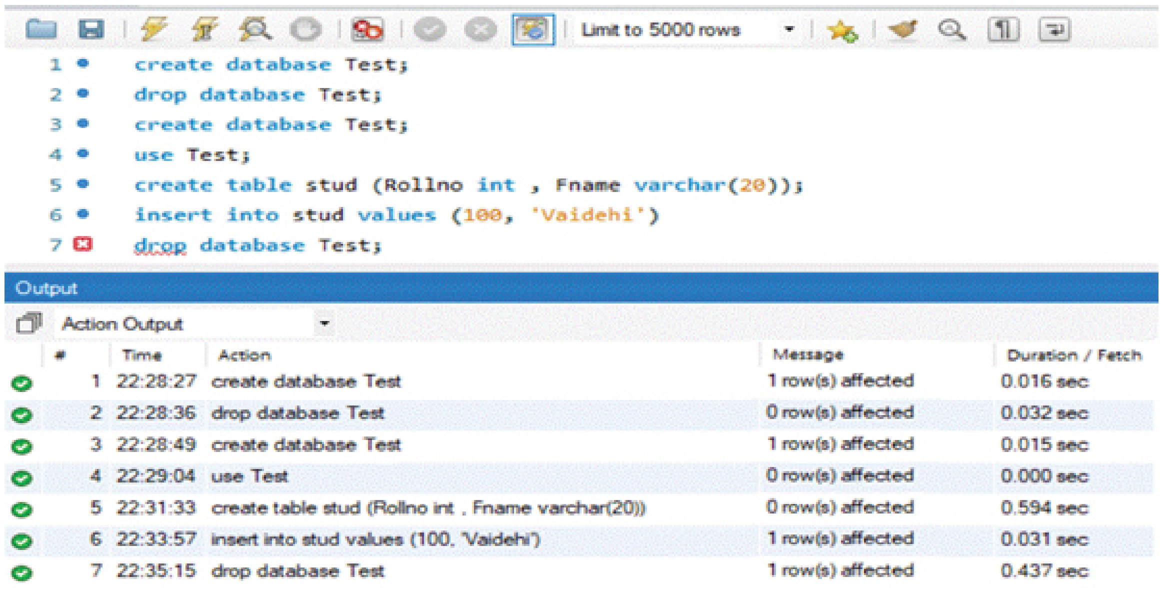
DROP DATABASE Test;

The preceding query will delete the complete database including all its tables with their data, as shown in *Figure 4.6*. Once the database is deleted, there is no way to get it back.

Similarly, the DROP commands can be used to drop an index and view, as follows:

- DROP view view-name
- DROP index index-name
- DROP constraint constraint_name

Refer to *Figure 4.6* that illustrates the `DROP` database command:



```
create database Test;
drop database Test;
create database Test;
use Test;
create table stud (Rollno int , Fname varchar(20));
insert into stud values (100, 'Vaidehi')
drop database Test;
```

Output

Action Output

```
 #  Time      Action                                         Message            Duration / Fetch
 1  22:28:27  create database Test                           1 row(s) affected  0.016 sec
 2  22:28:36  drop database Test                             0 row(s) affected  0.032 sec
 3  22:28:49  create database Test                           1 row(s) affected  0.015 sec
 4  22:29:04  use Test                                       0 row(s) affected  0.000 sec
 5  22:31:33  create table stud (Rollno int , Fname varchar(20))  0 row(s) affected  0.594 sec
 6  22:33:57  insert into stud values (100, 'Vaidehi')       1 row(s) affected  0.031 sec
 7  22:35:15  drop database Test                             1 row(s) affected  0.437 sec
```

Figure 4.6: *DROP DATABASE Command*

ALTER Command

Definition: It is used to alter the structure of the database. This alteration could be either to modify the definition of an existing attribute or delete the existing attributes, or it may be to add a new attribute.

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

Syntax:

ALTER TABLE table_name ADD (column-1 datatype, column-2 datatype, ….. column-an datatype);

Example: Consider the `STAFF` table, where there is a need to add one more column as `Sid_AC` (since the Staff coordinator required for each staff):

ALTER TABLE STAFF ADD Sid_AC int ;

The preceding query will add one column `Sid_AC` into the `STAFF` table and store the integer values.

To modify a column in a table, the following is the syntax:

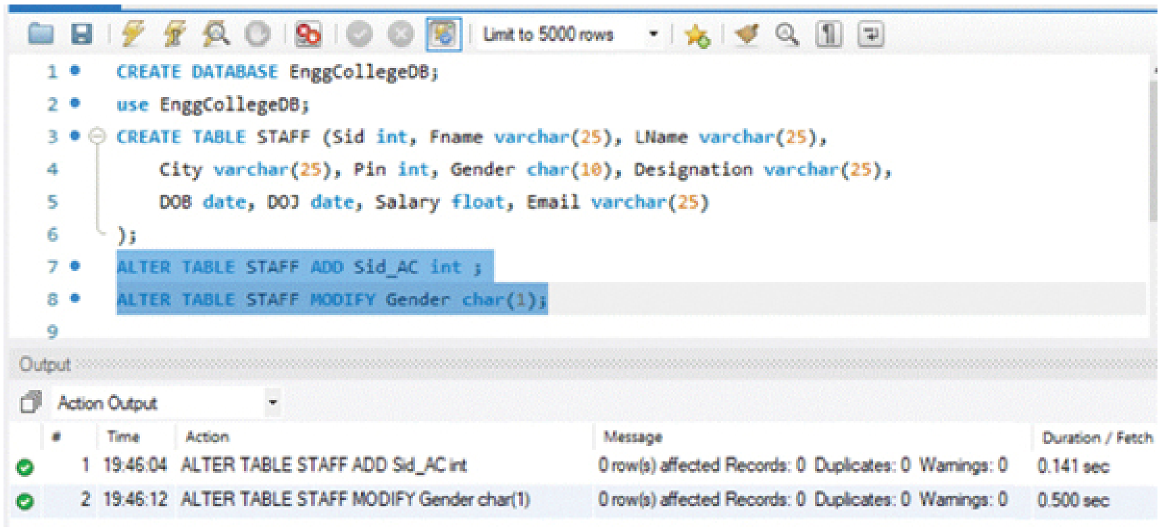
Syntax:

ALTER TABLE table_name MODIFY (column-1 datatype, column-2 datatype, ….. column-n datatype);

Example: Consider the `STAFF` table, where there is a need to redefine the datatype of the Gender column from char(10) to char(1):

ALTER TABLE STAFF MODIFY Gender char(1);

The preceding query will modify the Gender column to store the character values from 10 characters to only holding a single character, as depicted in *Figure 4.7*:



```
CREATE DATABASE EnggCollegeDB;
use EnggCollegeDB;
CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),
    City varchar(25), Pin int, Gender char(10), Designation varchar(25),
    DOB date, DOJ date, Salary float, Email varchar(25)
);
ALTER TABLE STAFF ADD Sid_AC int ;
ALTER TABLE STAFF MODIFY Gender char(1);
```

```
#  Time      Action                                  Message                                                       Duration / Fetch
1  19:46:04  ALTER TABLE STAFF ADD Sid_AC int        0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0     0.141 sec
2  19:46:12  ALTER TABLE STAFF MODIFY Gender char(1) 0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0     0.500 sec
```

***Figure 4.7:** ALTER Table Command*

To delete a column in a table, the following is the syntax:

Syntax: `ALTER TABLE DROP COLUMN column_name ;`

Example: Assume table Exam having columns Exam_ID, Exam_Date, Exam_Time, and Duration. Suppose we would like to delete the Duration column from the Exam table. The following query will fulfill the requirement:

ALTER TABLE DROP COLUMN Duration ;

TRUNCATE Command

Definition: It is used to delete all the rows from the table and make the table empty, though the structure will remain the same. It also releases the storage space used by that table. We cannot rollback the rows removed through the truncate command.

Syntax:

TRUNCATE TABLE table_name ;

Example: Assume that we want to delete an Exam table with 04 columns and 25000 records, then fire the following query:

TRUNCATE TABLE Exam ;

The preceding query will delete all 25000 records from the Exam table; however, it will maintain the structure of the Exam table.

RENAME Command

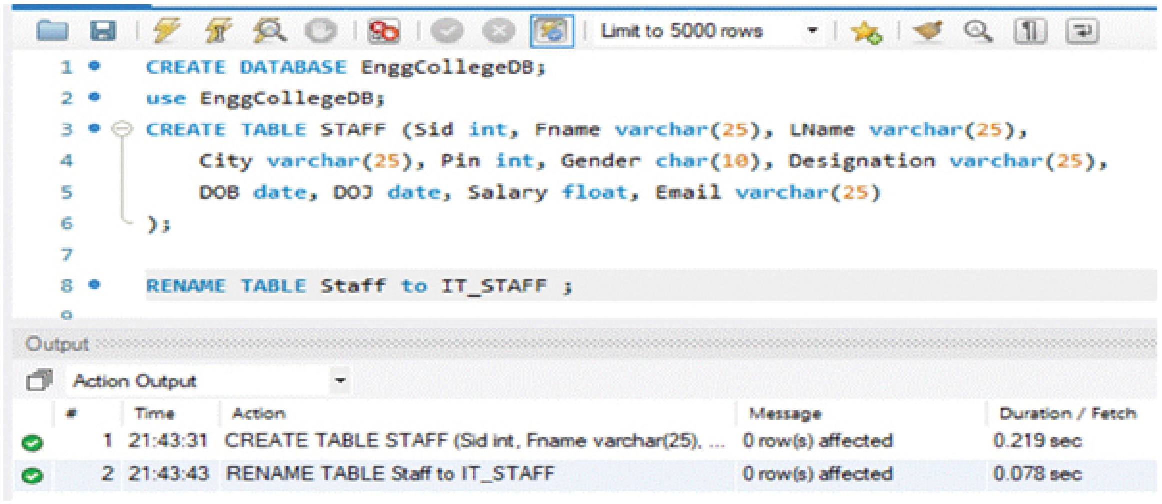
Definition: It is used to rename the database objects. It means that this command is used to change the name of a table, view, sequence, or synonym.

Syntax:

RENAME TABLE old_name to new_name;

Example: Suppose you want to change the name of the STAFF table as `IT_STAFF`, then following query can be fired to achieve the desired result, and it is depicted in *Figure 4.8*:

RENAME TABLE STAFF to IT_STAFF ;



```
CREATE DATABASE EnggCollegeDB;
use EnggCollegeDB;
CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),
    City varchar(25), Pin int, Gender char(10), Designation varchar(25),
    DOB date, DOJ date, Salary float, Email varchar(25)
);

RENAME TABLE Staff to IT_STAFF ;
```

```
Output
Action Output
   #  Time      Action                                           Message            Duration / Fetch
   1  21:43:31  CREATE TABLE STAFF (Sid int, Fname varchar(25), ...  0 row(s) affected  0.219 sec
   2  21:43:43  RENAME TABLE Staff to IT_STAFF                   0 row(s) affected  0.078 sec
```

Figure 4.8: *RENAME Table Command*

COMMENT Command

Definition: We can place the comments within the SQL statements. Comments can be used to document the purpose of an SQL statement or the logic of a code block in a stored procedure.

MySQL supports three comment styles, as shown in *Figure 4.9*:

Syntax:

1. From a '-- ' to the end of the line.

2. From a '#' to the end of the line.

Example:

`ALTER TABLE TEST ADD Test_Marks ;` -- adding one column to store the test marks.

Example:

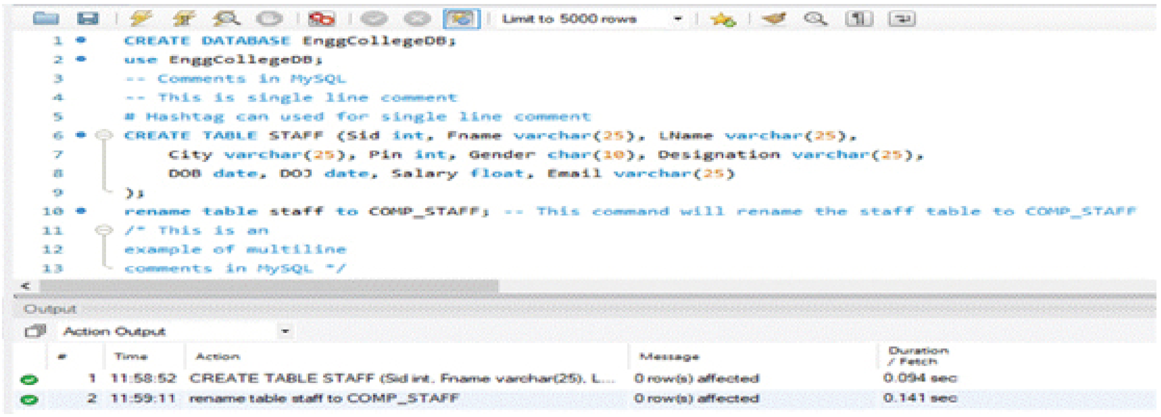
`ALTER TABLE TEST DROP column Test_Location ;` # deleted Test_Location column.

3. /**/ can span multiple lines. You use this comment style to document a block of SQL code.

Example:

/*

This is an example of multiline
Comments in MySQL*/



```
CREATE DATABASE EnggCollegeDB;
use EnggCollegeDB;
-- Comments in MySQL
-- This is single line comment
# Hashtag can used for single line comment
CREATE TABLE STAFF (Sid int, Fname varchar(25), LName varchar(25),
    City varchar(25), Pin int, Gender char(10), Designation varchar(25),
    DOB date, DOJ date, Salary float, Email varchar(25)
);
rename table staff to COMP_STAFF; -- This command will rename the staff table to COMP_STAFF
/* This is an
example of multiline
comments in MySQL */
```

```
   #  Time      Action                                           Message              Duration / Fetch
   1  11:58:52  CREATE TABLE STAFF (Sid int, Fname varchar(25), L...  0 row(s) affected  0.094 sec
   2  11:59:11  rename table staff to COMP_STAFF                 0 row(s) affected    0.141 sec
```

Figure 4.9: *COMMENTS in SQL*

DESC Command

Definition: The DESC is the short form of the DESCRIBE command and is used to display the information about a table, like its column names with their data types and what constraints are applied on these columns. This command is useful when we want to find out the structure of a table, and it displays the information of each column of a table.

Syntax:

DESC table_name;

Or

DESCRIBE table_name;

Example:

DESC STAFF;

Data Manipulation Language

The DML commands are the most frequently used SQL commands and are used to retrieve and manipulate the existing data in the database. It is further classified into Procedural and Non-Procedural DML.

The following commands come under DML, which are used to retrieve, store, update, and delete the data in a database:

- `INSERT` - Inserts new data into a table
- `SELECT` - Extracts the data from a database
- `UPDATE` - Updates the existing data within a table
- `DELETE` - Deletes the data from a table

INSERT Command

Definition: This is used to add one or more rows into a table.

There are two ways to write the `INSERT` statement, as depicted in *Figure 4.10*:

1. If you specify both the column names and the values for all the columns to be inserted in the SQL query, the following will be the syntax:

   Syntax:

   INSERT INTO table_name (column1, column2, column3, ...)

   VALUES (value1, value2, value3, ...);

   The syntax specifies that the values are separated by commas, and the values other than the numerical values must be enclosed in apostrophes such as char, varchar, date etc. The values must be entered in the same order as they are defined in the column list.

Example:

INSERT INTO DEPARTMENT (Did, Name, HOD_id, HOD_Start_Date)

VALUES(1, 'Computer Engineering', 101, '2018-7-11');

2. If you do not specify the column names and the value for all the columns needs to be inserted in the SQL query, the following is the syntax:

   Syntax:

   INSERT INTO table_name

   VALUES (value1, value2, value3, ...);

   Example:

   INSERT INTO DEPARTMENT

   VALUES(3, 'Electronics Engineering', 301, '2017-6-20');

3. To add values in specific columns or change the order in the SQL query, the following is the syntax:

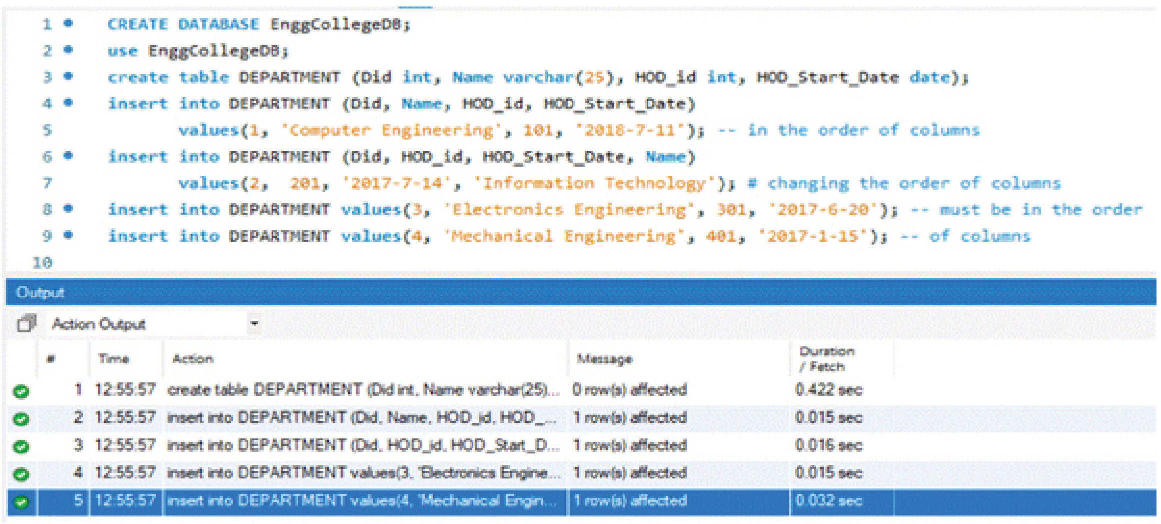
   Syntax:

   INSERT INTO table_name (column names to which data to be inserted separated by comma) VALUES (list of values separated by comma);

   Example:

   INSERT INTO DEPARTMENT (Did, HOD_id, HOD_Start_Date, Name)

   VALUES(2, 201, '2017-7-14', 'Information Technology');



```
CREATE DATABASE EnggCollegeDB;
use EnggCollegeDB;
create table DEPARTMENT (Did int, Name varchar(25), HOD_id int, HOD_Start_Date date);
insert into DEPARTMENT (Did, Name, HOD_id, HOD_Start_Date)
        values(1, 'Computer Engineering', 101, '2018-7-11'); -- in the order of columns
insert into DEPARTMENT (Did, HOD_id, HOD_Start_Date, Name)
        values(2,  201, '2017-7-14', 'Information Technology'); # changing the order of columns
insert into DEPARTMENT values(3, 'Electronics Engineering', 301, '2017-6-20'); -- must be in the order
insert into DEPARTMENT values(4, 'Mechanical Engineering', 401, '2017-1-15'); -- of columns
```

	#	Time	Action	Message	Duration / Fetch
✓	1	12:55:57	create table DEPARTMENT (Did int, Name varchar(25)...	0 row(s) affected	0.422 sec
✓	2	12:55:57	insert into DEPARTMENT (Did, Name, HOD_id, HOD_...	1 row(s) affected	0.015 sec
✓	3	12:55:57	insert into DEPARTMENT (Did, HOD_id, HOD_Start_D...	1 row(s) affected	0.016 sec
✓	4	12:55:57	insert into DEPARTMENT values(3, 'Electronics Engine...	1 row(s) affected	0.015 sec
✓	5	12:55:57	insert into DEPARTMENT values(4, 'Mechanical Engin...	1 row(s) affected	0.032 sec

Figure 4.10: Various ways of execution of INSERT command

SELECT Command

Definition: It is used to retrieve the data from a database. The simplified syntax of the `SELECT` command is as follows:

Syntax:

SELECT(column_list)

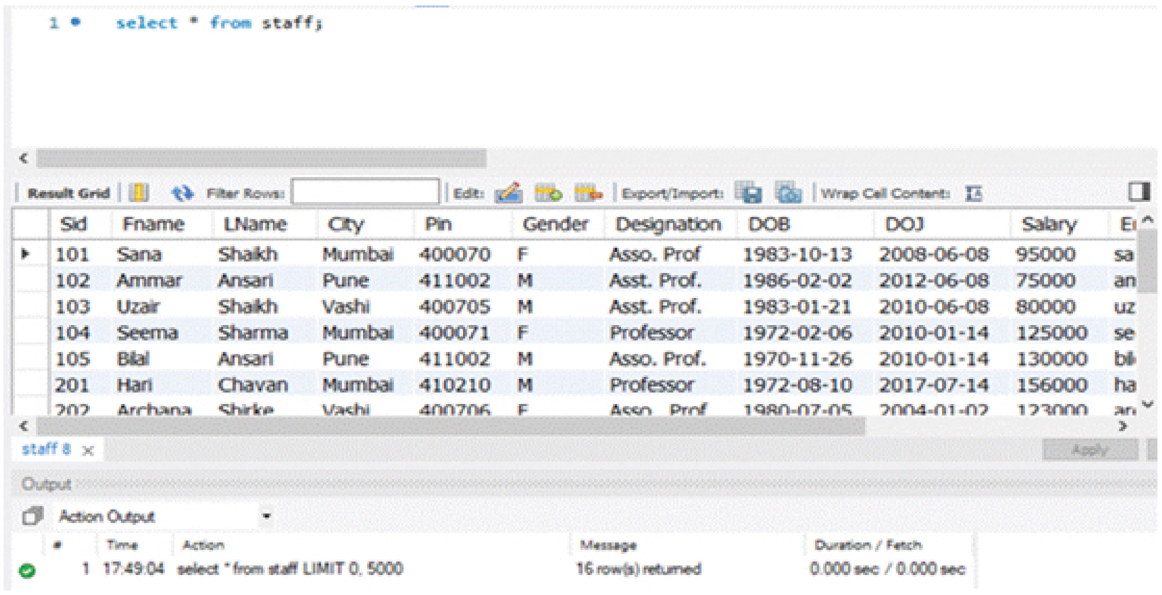
FROM table_name

[WHERE condition/expression]

[ORDER BY (column_list) [ASC | DESC]] ;

Example: When we want to retrieve all columns with all rows from a table, the following query can be fired:

`SELECT * FROM STAFF ;` -- '*' stands for all columns of the table, as shown in *Figure 4.11*:



```
select * from staff;
```

Sid	Fname	LName	City	Pin	Gender	Designation	DOB	DOJ	Salary	E
101	Sana	Shaikh	Mumbai	400070	F	Asso. Prof	1983-10-13	2008-06-08	95000	sa
102	Ammar	Ansari	Pune	411002	M	Asst. Prof.	1986-02-02	2012-06-08	75000	an
103	Uzair	Shaikh	Vashi	400705	M	Asst. Prof.	1983-01-21	2010-06-08	80000	uz
104	Seema	Sharma	Mumbai	400071	F	Professor	1972-02-06	2010-01-14	125000	se
105	Bilal	Ansari	Pune	411002	M	Asso. Prof.	1970-11-26	2010-01-14	130000	bi
201	Hari	Chavan	Mumbai	410210	M	Professor	1972-08-10	2017-07-14	156000	ha
202	Archana	Shirke	Vashi	400706	F	Asso. Prof	1980-07-05	2004-01-02	123000	ar

```
#  Time      Action                              Message              Duration / Fetch
1  17:49:04  select * from staff LIMIT 0, 5000   16 row(s) returned   0.000 sec / 0.000 sec
```

Figure 4.11: Basic SELECT command

Example: When we want to retrieve specific columns from a table, it can be listed in the `SELECT` clause as shown in the following command:

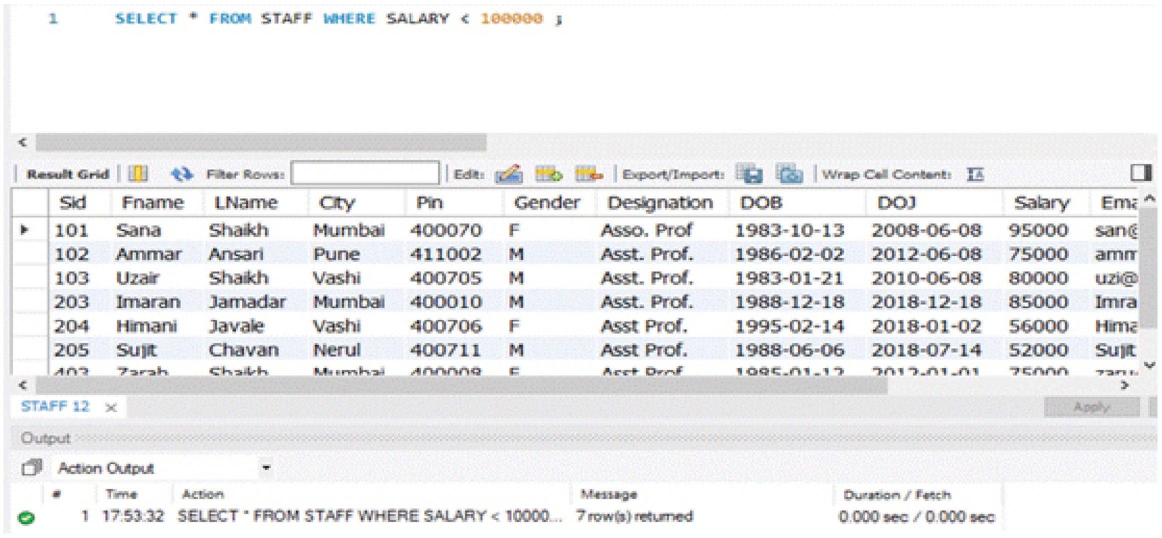
`SELECT Sid, Fname, Designation, Salary FROM STAFF ;` -- will retrieve Staff's ID, First Name, Designation and Salary.

Example: When we want to eliminate the duplicate values of the specific columns, the DISTINCT keyword can be used, as shown in the following query:

SELECT DISTINCT Designation FROM STAFF;

Example: Select command with `WHERE` clause, as shown in Figure 4.12:

SELECT * FROM STAFF WHERE SALARY < 100000 ;



```
SELECT * FROM STAFF WHERE SALARY < 100000 ;
```

Sid	Fname	LName	City	Pin	Gender	Designation	DOB	DOJ	Salary	Emi
101	Sana	Shaikh	Mumbai	400070	F	Asso. Prof	1983-10-13	2008-06-08	95000	san@
102	Ammar	Ansari	Pune	411002	M	Asst. Prof.	1986-02-02	2012-06-08	75000	amm
103	Uzair	Shaikh	Vashi	400705	M	Asst. Prof.	1983-01-21	2010-06-08	80000	uzi@
203	Imaran	Jamadar	Mumbai	400010	M	Asst. Prof.	1988-12-18	2018-12-18	85000	Imra
204	Himani	Javale	Vashi	400706	F	Asst Prof.	1995-02-14	2018-01-02	56000	Hima
205	Sujit	Chavan	Nerul	400711	M	Asst Prof.	1988-06-06	2018-07-14	52000	Sujit
403	Zarah	Shaikh	Mumbai	400008	F	Asst Prof	1985-01-12	2012-01-01	75000	zar

STAFF 12

Output

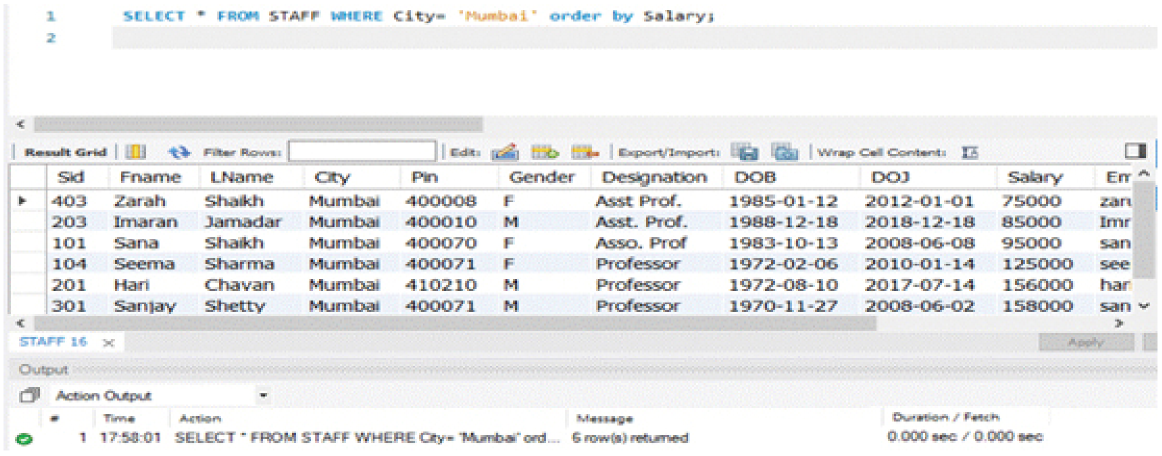
Action Output

```
#  Time      Action                                      Message             Duration / Fetch
1  17:53:32  SELECT * FROM STAFF WHERE SALARY < 10000...  7 row(s) returned   0.000 sec / 0.000 sec
```

Figure 4.12: SELECT command with WHERE clause

Example: Select command with `order by` clause, as shown in Figure *4.13*:

SELECT * FROM STAFF WHERE City= "Mumbai" order by Salary;



```
SELECT * FROM STAFF WHERE City= 'Mumbai' order by Salary;
```

Sid	Fname	LName	City	Pin	Gender	Designation	DOB	DOJ	Salary	Em
403	Zarah	Shaikh	Mumbai	400008	F	Asst Prof.	1985-01-12	2012-01-01	75000	zar
203	Imaran	Jamadar	Mumbai	400010	M	Asst. Prof.	1988-12-18	2018-12-18	85000	Imr
101	Sana	Shaikh	Mumbai	400070	F	Asso. Prof	1983-10-13	2008-06-08	95000	san
104	Seema	Sharma	Mumbai	400071	F	Professor	1972-02-06	2010-01-14	125000	see
201	Hari	Chavan	Mumbai	410210	M	Professor	1972-08-10	2017-07-14	156000	har
301	Sanjay	Shetty	Mumbai	400071	M	Professor	1970-11-27	2008-06-02	158000	san

STAFF 16

Output

Action Output

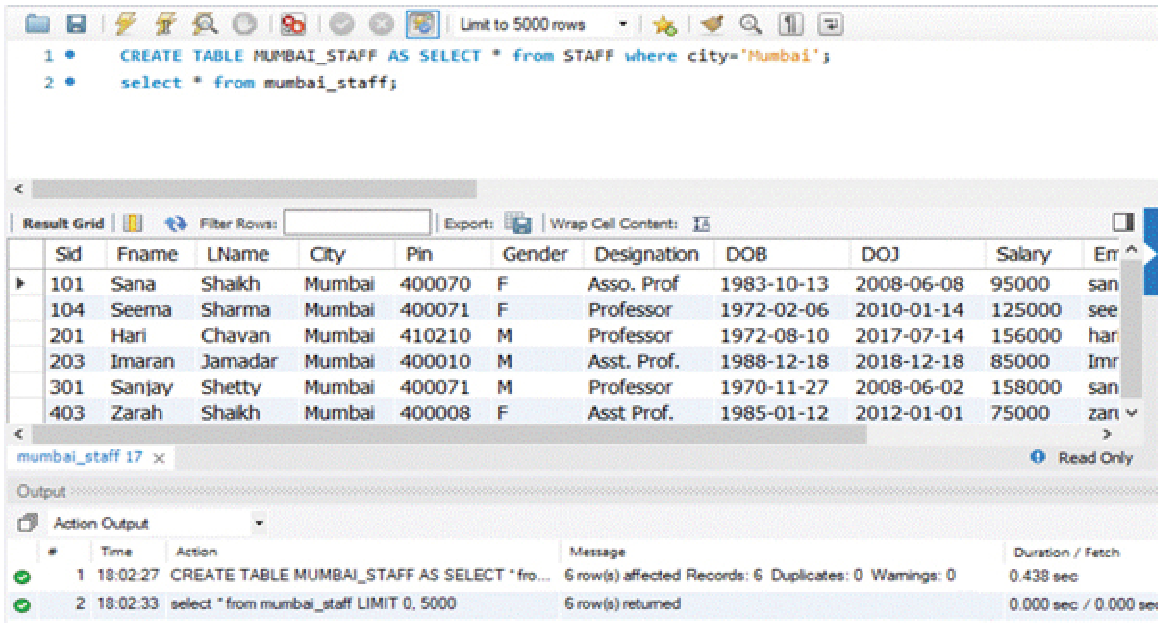
```
#  Time      Action                                          Message             Duration / Fetch
1  17:58:01  SELECT * FROM STAFF WHERE City= 'Mumbai' ord...  6 row(s) returned   0.000 sec / 0.000 sec
```

Figure 4.13: SELECT command with ORDER BY clause

Example: Select command to create a table, as shown in *Figure 4.14*:

Syntax: `CREATE TABLE table_name AS SELECT * from existing_table_name;`

Example: `CREATE TABLE MUMBAI_STAFF AS SELECT * from STAFF where city='Mumbai';`



Limit to 5000 rows

```
CREATE TABLE MUMBAI_STAFF AS SELECT * from STAFF where city='Mumbai';
select * from mumbai_staff;
```

Sid	Fname	LName	City	Pin	Gender	Designation	DOB	DOJ	Salary	Em
101	Sana	Shaikh	Mumbai	400070	F	Asso. Prof	1983-10-13	2008-06-08	95000	san
104	Seema	Sharma	Mumbai	400071	F	Professor	1972-02-06	2010-01-14	125000	see
201	Hari	Chavan	Mumbai	410210	M	Professor	1972-08-10	2017-07-14	156000	har
203	Imaran	Jamadar	Mumbai	400010	M	Asst. Prof.	1988-12-18	2018-12-18	85000	Imr
301	Sanjay	Shetty	Mumbai	400071	M	Professor	1970-11-27	2008-06-02	158000	san
403	Zarah	Shaikh	Mumbai	400008	F	Asst Prof.	1985-01-12	2012-01-01	75000	zar

mumbai_staff 17 — Read Only

Output — Action Output

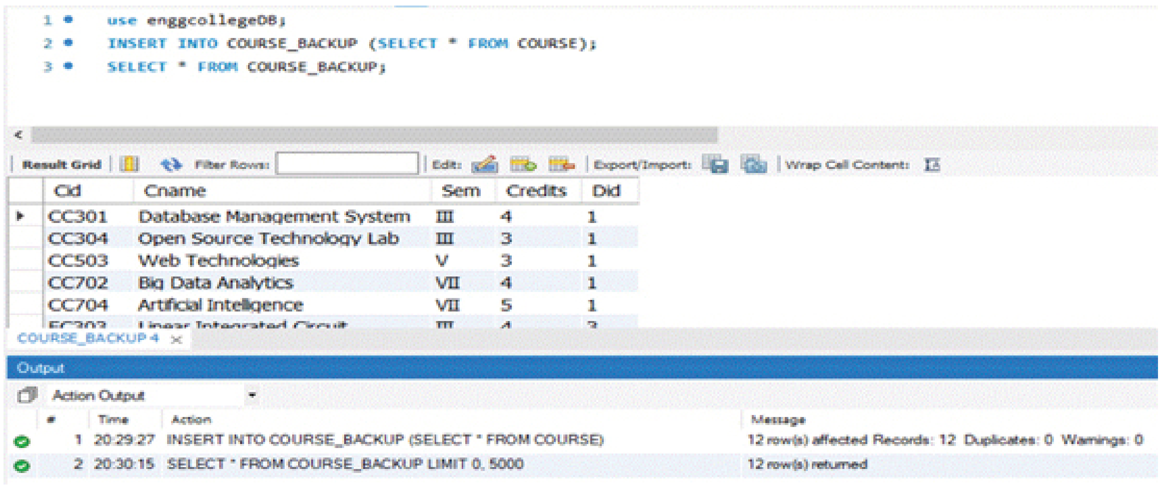
```
#  Time      Action                                        Message                                                   Duration / Fetch
1  18:02:27  CREATE TABLE MUMBAI_STAFF AS SELECT * fro...  6 row(s) affected Records: 6  Duplicates: 0  Warnings: 0  0.438 sec
2  18:02:33  select * from mumbai_staff LIMIT 0, 5000      6 row(s) returned                                         0.000 sec / 0.000 sec
```

Figure 4.14: *Create Table using SELECT command*

Example: Select command to insert the records, as shown in Figure 4.15:

Syntax: `INSERT INTO table_name ( SELECT columns from existing_table_name);`

Example: `INSERT INTO COURSE_BACKUP (SELECT * FROM COURSE);`



```
use enggcollegeDB;
INSERT INTO COURSE_BACKUP (SELECT * FROM COURSE);
SELECT * FROM COURSE_BACKUP;
```

Cid	Cname	Sem	Credits	Did
CC301	Database Management System	III	4	1
CC304	Open Source Technology Lab	III	3	1
CC503	Web Technologies	V	3	1
CC702	Big Data Analytics	VII	4	1
CC704	Artificial Intelligence	VII	5	1
EC303	Linear Integrated Circuit	III	4	3

COURSE_BACKUP 4

Output — Action Output

```
#  Time      Action                                        Message
1  20:29:27  INSERT INTO COURSE_BACKUP (SELECT * FROM COURSE)  12 row(s) affected Records: 12  Duplicates: 0  Warnings: 0
2  20:30:15  SELECT * FROM COURSE_BACKUP LIMIT 0, 5000         12 row(s) returned
```

Figure 4.15: *Insert records using SELECT command*

UPDATE Command

Definition: It is used to modify the existing data in a table.

Syntax:

UPDATE table_name

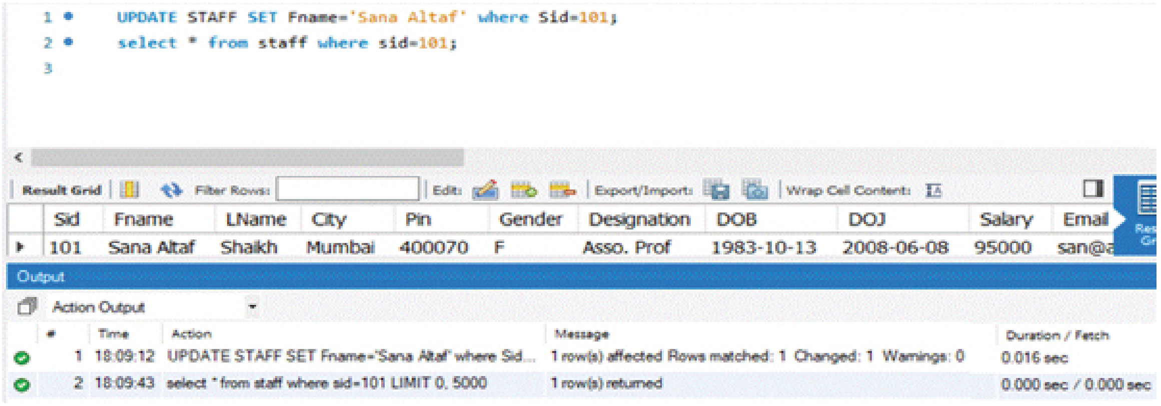
SET column1 = value1, column2 = value2, ...

[WHERE condition];

Example:

UPDATE STAFF SET Fname='Sana Altaf' where Sid=101;

The preceding query will update the Staff member's Fname from "Sana" to "Sana Altaf " whose Sid is 101. If we do not use the where condition in the preceding query, then it will update all the staff member's Fname as 'Sana Altaf, as shown in *Figure 4.16*:



```
UPDATE STAFF SET Fname='Sana Altaf' where Sid=101;
select * from staff where sid=101;
```

Sid	Fname	LName	City	Pin	Gender	Designation	DOB	DOJ	Salary	Email
101	Sana Altaf	Shaikh	Mumbai	400070	F	Asso. Prof	1983-10-13	2008-06-08	95000	san@a

#	Time	Action	Message	Duration / Fetch
1	18:09:12	UPDATE STAFF SET Fname='Sana Altaf' where Sid...	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.016 sec
2	18:09:43	select * from staff where sid=101 LIMIT 0, 5000	1 row(s) returned	0.000 sec / 0.000 sec

Figure 4.16: *UPDATE command*

DELETE Command

Definition: The `DELETE` command is used to delete the rows from the tables. This command has an optional where clause. The `WHERE` clause specifies which record or records needs to be deleted from the database. If the `WHERE` clause is not specified, it will delete all the records and make the table empty.

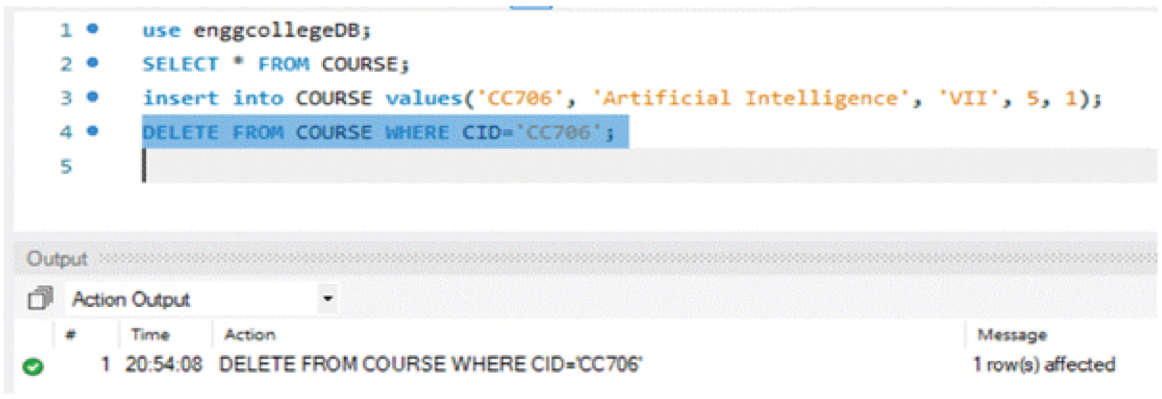
Syntax:

DELETE FROM table_name [WHERE condition] ;

Example:

DELETE FROM COURSE WHERE Cid = ‘CC706’ ;

The preceding query will delete the record from the COURSE table whose course ID is equal to ‘CC706’, as depicted in *Figure 4.17*:



```
use enggcollegeDB;
SELECT * FROM COURSE;
insert into COURSE values('CC706', 'Artificial Intelligence', 'VII', 5, 1);
DELETE FROM COURSE WHERE CID='CC706';
```

```
#   Time      Action                                    Message
1   20:54:08  DELETE FROM COURSE WHERE CID='CC706'      1 row(s) affected
```

Figure 4.17: *DELETE Command with WHERE clause*

Data Control Language

It mainly deals with the rights, permissions, and other controls of the database system. Since, there are different types of users and it depends on the user’s role and responsibilities, DBA is supposed to assign the rights. The rights that allow the users to use some or all objects of the database are called **privileges**. When a user creates any objects in the database, it will become the owner and have complete control over them. If any user wants to access any of the objects of another user, the owner needs to grant the privileges. After giving privileges, the owner can take it back by revoking the privileges.

The SQL data-control language includes the commands to grant and revoke the privileges. The SQL standard includes select, insert, delete, and update privileges. The select privilege corresponds to the read privilege. SQL includes a references privilege which restricts a user’s ability to declare foreign keys while creating a relationship between the parent and child relations. If the relation to be created includes a foreign key that references the attributes of another relation, the user must have been granted the reference privilege on those attributes.

GRANT Command

Definition: The grant statement is used to confer authorization, i.e., it is used to give access permission to the users for accessing the database objects.

Syntax: The basic form of the GRANT statement is as follows :

```
GRANT object_privileges
ON object_name
TO User_Name
[WITH GRANT OPTION]
```

Example:

```
GRANT ALL
ON COURSE TO Zahra
WITH GRANT OPTION
```

In the preceding query, WITH GRANT OPTION allows the users to further grant object privileges to other users. Here, Zahara has all the privileges on the COURSE table having WITH GRANT OPTION. Assume Zahra wants to give only SELECT and UPDATE permission rights on the COURSE table to Sam. At the same time, she does not want him to further give permission to others. To achieve this, the following query can be fired:

```
GRANT SELECT, UPDATE ON COURSE TO Sam;
```

REVOKE Command

The Revoke command has identical form as that of the Grant command and is used to revoke an authorization. In certain cases, the revoke command may have a cascading impact and can be avoided by using restrict.

Definition: It is used to return the assigned access permission from the users.

Syntax:

```
REVOKE Object_Privileges
ON Object_Name
FROM User_Name
```

Example:

```
REVOKE SELECT, UPDATE
ON COURSE
FROM Sam;
```

Transaction Control Language

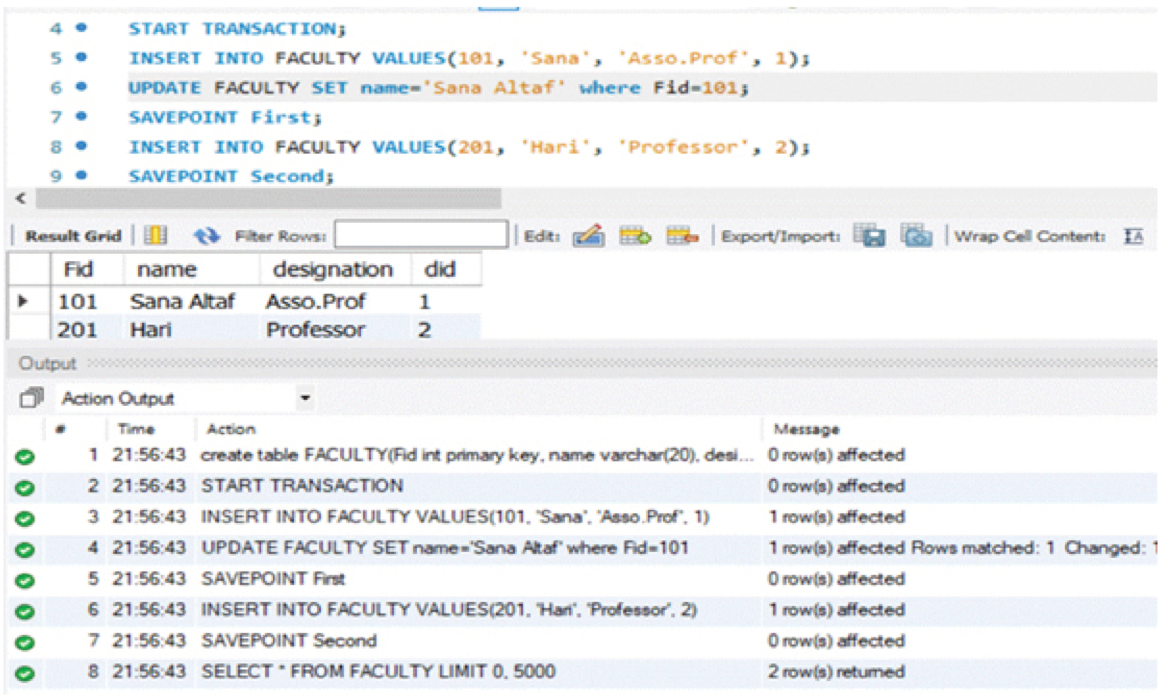
To perform and manage the transactions in the database, the Transaction Control Language commands are used. The **COMMIT, ROLLBACK**, and **SAVEPOINT** statements are also called the **transaction control language (TCL)** statements. These commands manage the changes made by the DML-statements.

COMMIT: The Commit command is used to permanently save the changes done by the user into the database. In logical transactions, the final command issued is the commit command.

The following MySQL code demonstrate the TCL commands:

```
use enggcollegeDB;
create table FACULTY(Fid int primary key, name varchar(20), designation
varchar(20), did int references DEPARTMENT);
START TRANSACTION;
INSERT INTO FACULTY VALUES(101, 'Sana', 'Asso.Prof', 1);
UPDATE FACULTY SET name='Sana Altaf' where Fid=101;
SAVEPOINT First;
INSERT INTO FACULTY VALUES(201, 'Hari', 'Professor', 2);
SAVEPOINT Second;
SELECT * FROM FACULTY;
ROLLBACK TO First;
SELECT * FROM FACULTY;
```

Refer to *Figure 4.18* that illustrates the TCL – Savepoint command:



```
START TRANSACTION;
INSERT INTO FACULTY VALUES(101, 'Sana', 'Asso.Prof', 1);
UPDATE FACULTY SET name='Sana Altaf' where Fid=101;
SAVEPOINT First;
INSERT INTO FACULTY VALUES(201, 'Hari', 'Professor', 2);
SAVEPOINT Second;
```

Fid	name	designation	did
101	Sana Altaf	Asso.Prof	1
201	Hari	Professor	2

```
#  Time      Action                                                          Message
1  21:56:43  create table FACULTY(Fid int primary key, name varchar(20), desi...  0 row(s) affected
2  21:56:43  START TRANSACTION                                               0 row(s) affected
3  21:56:43  INSERT INTO FACULTY VALUES(101, 'Sana', 'Asso.Prof', 1)         1 row(s) affected
4  21:56:43  UPDATE FACULTY SET name='Sana Altaf' where Fid=101              1 row(s) affected Rows matched: 1  Changed: 1
5  21:56:43  SAVEPOINT First                                                 0 row(s) affected
6  21:56:43  INSERT INTO FACULTY VALUES(201, 'Hari', 'Professor', 2)         1 row(s) affected
7  21:56:43  SAVEPOINT Second                                                0 row(s) affected
8  21:56:43  SELECT * FROM FACULTY LIMIT 0, 5000                             2 row(s) returned
```

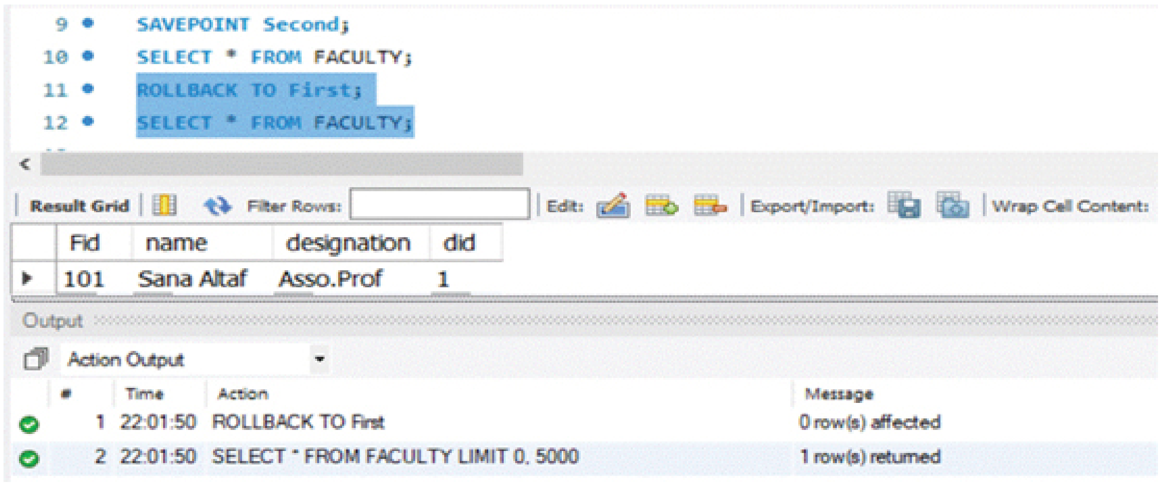
Figure 4.18: *TCL - Savepoint Command*

ROLLBACK

The Rollback command is just the opposite of the Commit command. It restores the database to the last committed state. Rollback performs the undo operation which was carried out during the logical transaction processing. It undoes the operations till the last commit command.

SAVEPOINT

The Transaction Control Language (TCL) commands manage the logical transactions in the database. The DML statements make the changes to the data. A set of DML statements/commands form a logical transaction, wherein the SAVEPOINT command is used to save the transaction temporarily. In case of rollback, changes done till SAVEPOINT will be unchanged, that is, the transaction can be rolled back to that point whenever required. *Figure 4.18* shows the execution of the transaction with the savepoint command and *Figure 4.19* shows the execution of the rollback:



```
SAVEPOINT Second;
SELECT * FROM FACULTY;
ROLLBACK TO First;
SELECT * FROM FACULTY;
```

Fid	name	designation	did
101	Sana Altaf	Asso.Prof	1

```
#  Time      Action                              Message
1  22:01:50  ROLLBACK TO First                   0 row(s) affected
2  22:01:50  SELECT * FROM FACULTY LIMIT 0, 5000 1 row(s) returned
```

Figure 4.19: *TCL - Rollback Command*

SET Operations

Queries consisting of the set operators are known as compound queries. The Set operation is used to compare rows from two or more tables and then combine it. For many complex problems, it is much easier to use a set operation, compare it, and then join the data from two or more tables.

For applying the SET operations on relations A and B, both must be type compatible (or union compatible). Type compatible means, both relations A and B should have an equal number of attributes and also each corresponding pair of attributes should have the same domain.

The result of the union operation is a relation with the same schema as that input operand relations/tables.

The following are the types of Set Operation:

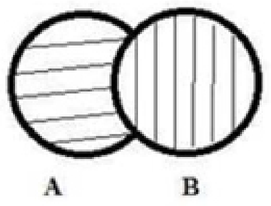
- Union
- UnionAll
- Intersect
- Minus

Union Operation

The result of A ∪ B would be a relation which includes all the tuples that are in A or in B or in both. It eliminates duplicate tuples, as shown as follows:

Syntax

```
SELECT column_name FROM table_name1
UNION
SELECT column_name FROM table_name2;
```

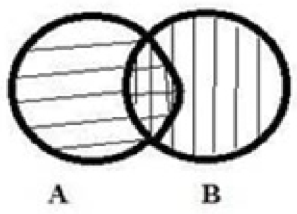


Union All Operation

The result of A ∪ B would be a relation which includes all the tuples that are in A or in B or in both. It includes duplicate tuples as well, as shown as follows:

Syntax:

```
SELECT column_name FROM table_name1
UNION ALL
SELECT column_name FROM table_name2;
```



Example:

Faculty Table

Fname	Lname
Bushra	Shaikh
Tabassum	Ansari
Siraj	Shaikh

Student Table

Fname	Lname
Sarah	Khan
Bushra	Shaikh
Rosy	D'souza
Joy	Pinto

Union

```
SELECT * FROM Faculty
UNION
SELECT * FROM Student;
```

OUTPUT

Fname	Lname
Bushra	Shaikh
Tabassum	Ansari
Siraj	Shaikh
Sarah	Khan
Rosy	D'souza
Joy	Pinto

6 rows

Union All

```
SELECT * FROM Faculty
UNION ALL
SELECT * FROM Student;
```

OUTPUT

Fname	Lname
Bushra	Shaikh
Tabassum	Ansari
Siraj	Shaikh
Sarah	Khan
Bushra	Shaikh
Rosy	D'souza
Joy	Pinto

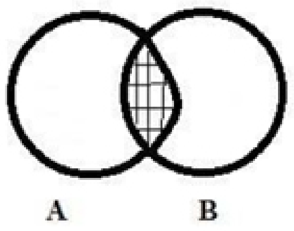
7 rows

Intersect Operation

The result of A ∩ B would be a relation which includes all the tuples that appear in both the relations A and B. It eliminates duplicate tuples, as shown as follows:

Syntax

```
SELECT column_name FROM table_name1
INTERSECT
SELECT column_name FROM table_name2;
```



Example:

INTERSECT

```
SELECT * FROM Faculty
INTERSECT
SELECT * FROM Student;
```

OUTPUT

Fname	Lname
Bushra	Shaikh

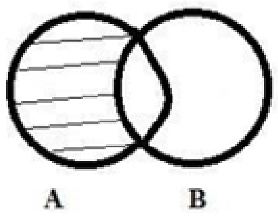
1 row

Minus Operation

The result of this operation, denoted by A – B, is a relation that includes all tuples that are in A relation but not in B relation, as shown as follows:

Syntax

```
SELECT column_name FROM table_name1
MINUS
SELECT column_name FROM table_name2;
```



Example:

MINUS

```
SELECT * FROM Faculty
MINUS
SELECT * FROM Student;
```

OUTPUT

Fname	Lname
Tabassum	Ansari
Siraj	Shaikh

2 rows

String operations

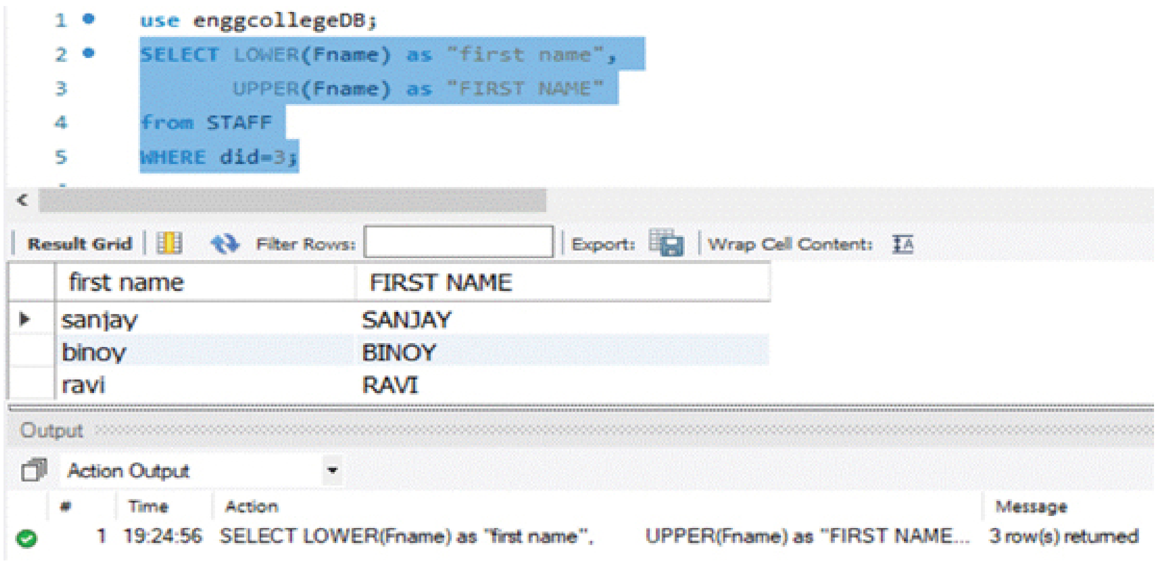
String functions are used to perform an operation on the input string and return an output string. String operators are very useful for many reasons, such as the following:

- To concat the data from multiple sources: For example, when we want to retrieve the complete name from the database, you may require to concat first name and last name.
- To Present the data in the required format.
- For case folding.
- Removal of white space from the data.
- Identifying the length of the string.
- Extracting the required substring from the data.
- Comparing the strings: For example, when there is a need to display the sorted list inherently, it uses the string comparisons.

I Case Manipulation Functions: These functions convert the case for the character strings, as shown in *Figure 4.20*:

Syntax:	**Example:**

SELECT LOWER(Column_name) from table_name;	SELECT LOWER(Fname) from STAFF;
Syntax: SELECT UPPER(Column_name) from table_name;	**Example:** SELECT UPPER(Lname) from STAFF;
Syntax: SELECT INITCAP(Column_name) from table_name;	**Example:** SELECT INITCAP(Dname) from DEPARTMENT;



```
use enggcollegeDB;
SELECT LOWER(Fname) as "first name",
       UPPER(Fname) as "FIRST NAME"
from STAFF
WHERE did=3;
```

```
first name    FIRST NAME
sanjay        SANJAY
binoy         BINOY
ravi          RAVI
```

```
#   Time      Action                                                           Message
1   19:24:56  SELECT LOWER(Fname) as "first name",   UPPER(Fname) as "FIRST NAME...   3 row(s) returned
```

Figure 4.20: *STRING - Case Manipulation Functions*

II Character manipulation functions: These functions manipulate the character strings as shown in *Figure 4.21*:

Syntax: SELECT CONCAT(Column1, Column2, Column3, … ..) from table_name;	**Example**: SELECT CONCAT(Fname, Lname) from STAFF;
Syntax: SELECT SUBSTR(string, start, length) from table_name;	**Example**: SELECT SUBSTR(Fname, 1, 3) from STAFF; -- extract first 3 characters from string
Syntax:	**Example**:

SELECT LENGTH(string) from table_name;	SELECT LENGTH(Fname) from STAFF;
Syntax: SELECT INSTR(string1, string2) from table_name;	**Example**: SELECT INSTR(Fname, “an”) from STAFF; -- Search for "an" in Fname column of STAFF table

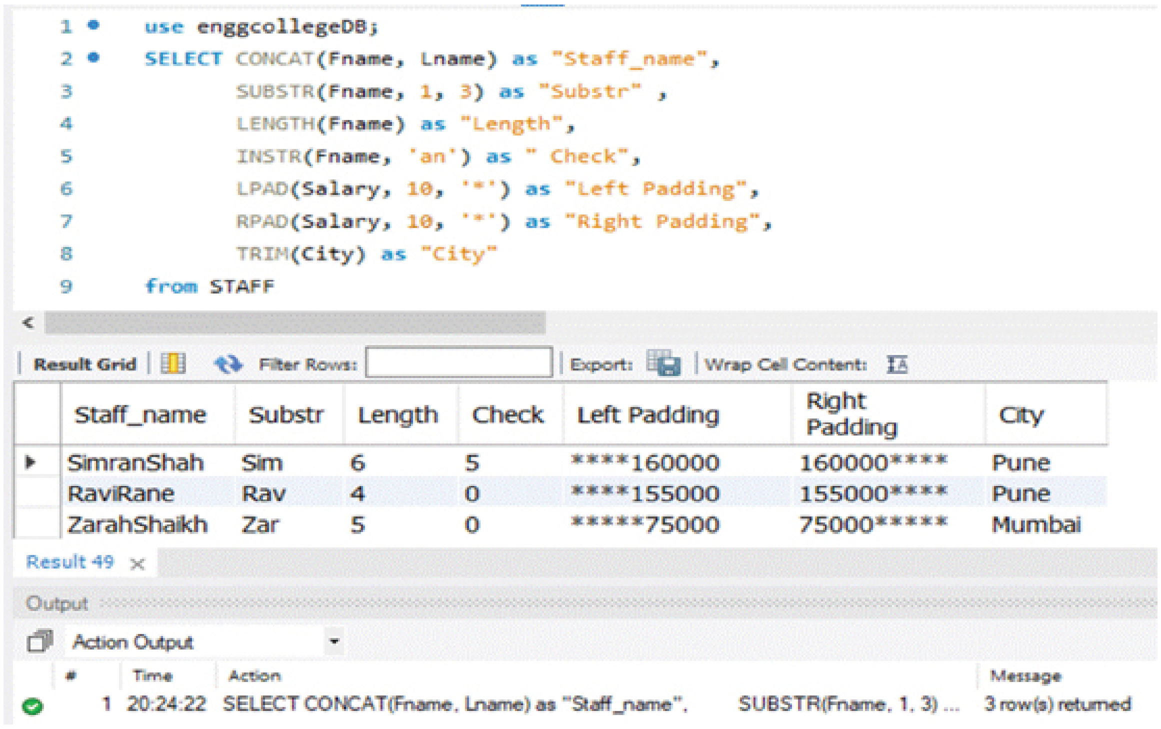
Syntax:
SELECT LPAD(string, length, lpad_string)
from table_name;
	Example:
	SELECT LPAD(Salary, 10, “*”)
	from STAFF;
	--Left-pad the text in "Salary" with "*", to a total length of 10
	Example: *****95000

Syntax:
SELECT RPAD(string, length, lpad_string)
from table_name;
	Example:
	SELECT RPAD(Salary, 10, “*”)
	from STAFF; --Right-pad the text in "Salary" with "*", to a total length of 10
	Example: 95000*****

Syntax: SELECT REPLACE(string, from_string, new_string) from table_name;	**Example**: SELECT REPLACE(Cname, “course”, “subject”) from COURSE;
Syntax: SELECT TRIM(string) from table_name;	**Example**: SELECT TRIM(City) from STAFF; -- removes leading and trailing spaces from City name

The following is the MySQL Code:

```
SELECT CONCAT(Fname, Lname) as "Staff_name",
      SUBSTR(Fname, 1, 3) as "Substr" ,
      LENGTH(Fname) as "Length",
      INSTR(Fname, 'an') as " Check",
      LPAD(Salary, 10, '*') as "Left Padding",
    RPAD(Salary, 10, '*') as "Right Padding",
    TRIM(City) as "City"
from STAFF
where DID=4;
```



```
use enggcollegeDB;
SELECT CONCAT(Fname, Lname) as "Staff_name",
        SUBSTR(Fname, 1, 3) as "Substr" ,
        LENGTH(Fname) as "Length",
        INSTR(Fname, 'an') as " Check",
        LPAD(Salary, 10, '*') as "Left Padding",
        RPAD(Salary, 10, '*') as "Right Padding",
        TRIM(City) as "City"
from STAFF
```

Staff_name	Substr	Length	Check	Left Padding	Right Padding	City
SimranShah	Sim	6	5	****160000	160000****	Pune
RaviRane	Rav	4	0	****155000	155000****	Pune
ZarahShaikh	Zar	5	0	*****75000	75000*****	Mumbai

```
#  Time      Action                                                              Message
1  20:24:22  SELECT CONCAT(Fname, Lname) as "Staff_name",   SUBSTR(Fname, 1, 3) ...  3 row(s) returned
```

Figure 4.21: *STRING - Character Manipulation Functions*

Aggregate functions, Group by, and Having

Considering from the database management perspective, the various organizations/enterprises have different requirements. Especially, the top management is normally more interested in the summarized report than that of the individual records. For example, at the end of the day, the sales manager is interested in the total sale. One can satisfy these types of requirements using the aggregate functions.

Aggregate Functions

An aggregate function allows us to easily produce the summarized data from the database and return a single value. Aggregate functions are often used with the `GROUP BY` clause of the `SELECT` statement. The most common five aggregate functions of SQL are as follows:

- COUNT: Counts rows in a specified table or view.
- MIN: Gets the minimum value in a set of values.
- MAX: Gets the maximum value in a set of values.
- SUM: Computes the sum of values.
- AVG: Computes the average of a set of values.

If we use an aggregate function without the `GROUP BY` clause in the SQL query, then it is equivalent to grouping on all the rows of the table. Unless otherwise stated, the aggregate functions ignore the NULL values. But the count aggregate function returns the number of tuples present in a table including the tuples having the 'Null' values. The `SUM()` and `AVG()` functions expect a numeric argument or cast the argument to a number, if necessary.

The `AVG()` returns an average of all the tuple values from an attribute of a table. The average aggregate function does not include 'Null value' while computing the average.

The simplified syntax of `AVG()` is as follows:
SELECT AVG ([ALL | DISTINCT] columname)
FROM TABLENAME WHERE CONDITION;

Example: For EnggCollegeDB, we can fire the following queries to find the average salaries of the staff:

# Compute the average salary of all staff

Select avg(SALARY) from STAFF;

# Compute Department wise average salary of all staff

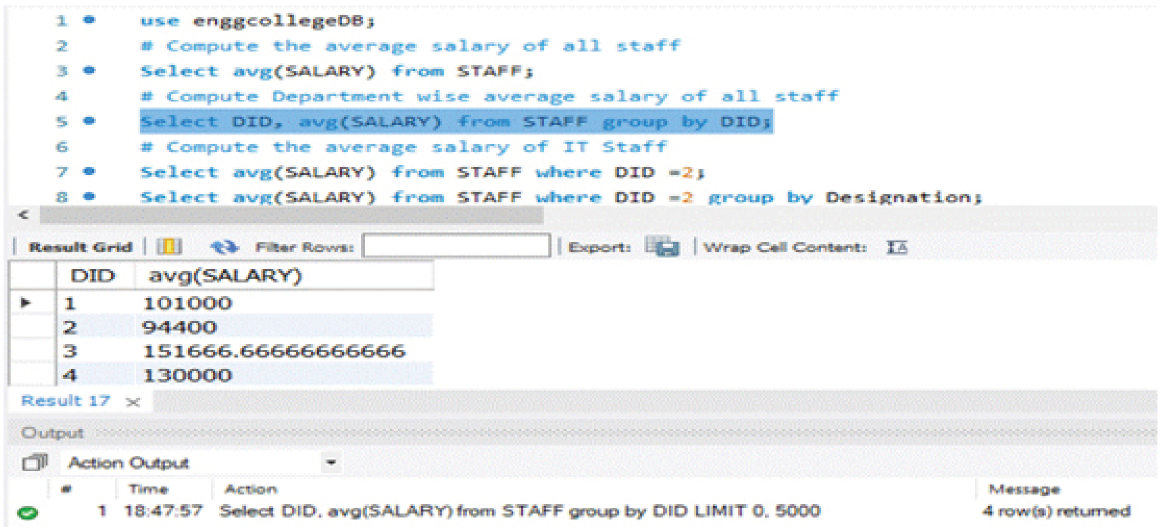
Select DID, avg(SALARY) from STAFF group by DID;

# Compute the average salary of IT Staff

Select avg(SALARY) from STAFF where DID =2;

Select avg(SALARY) from STAFF where DID =2 group by Designation;

Figure 4.22 shows the average function on the STAFF table of EnggCollegeDB to compute the average salary of the staff:



```
use enggcollegeDB;
# Compute the average salary of all staff
Select avg(SALARY) from STAFF;
# Compute Department wise average salary of all staff
Select DID, avg(SALARY) from STAFF group by DID;
# Compute the average salary of IT Staff
Select avg(SALARY) from STAFF where DID =2;
Select avg(SALARY) from STAFF where DID =2 group by Designation;
```

DID	avg(SALARY)
1	101000
2	94400
3	151666.66666666666
4	130000

```
#  Time      Action                                                    Message
1  18:47:57  Select DID, avg(SALARY) from STAFF group by DID LIMIT 0, 5000   4 row(s) returned
```

Figure 4.22: *Aggregate Function - Average*

COUNT

Aggregate function `COUNT` helps summarize the number of rows or tuples in the table. If no criterium is specified, it will return all rows or tuples, otherwise it will return the rows or tuples based on certain criteria specified in the where clause. It can work on both the numeric and the non-numeric data types. `COUNT` is the only aggregate function where we can use the "*", in the rest of the functions, we require an attribute or expression/formula for execution. The `COUNT(*)` considers duplicate and Null.

Syntax:

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

Example:

SELECT COUNT(*) FROM STAFF;

SELECT COUNT(*) FROM STAFF WHERE DID=2;

SELECT COUNT(DISTINCT DID) FROM STAFF;

Minimum (min)

The aggregate function min evaluates the minimum value among all the row/tuple values of the specified attribute. It does not include the 'null values' as an input for computation of the minimum value for a certain column.

Syntax:

`MIN()` or `MIN( [ALL|DISTINCT] expression )`

Example:

For EnggCollegeDB to find the minimum salary of all staff members, the query will be as follows:

SELECT MIN(salary) FROM STAFF;

To find the minimum salary from the Information Technology Department, fire the following query:

SELECT MIN(salary) FROM STAFF WHERE DID=2;

Maximum (max)

The max function is used to find the maximum/largest value of a column among all the tuple values. Like the other aggregate functions, it does not use 'null values' for the computation of the largest value from the column.

Syntax:

`MAX()` or `MAX ( [ALL|DISTINCT] expression )`

Example:

For EnggCollegeDB to find the Maximum salary of all staff members, the query will be as follows:

SELECT MAX (salary) FROM STAFF;

To find the maximum salary from the Computer Engineering Department, fire the following query:

SELECT MAX (salary) FROM STAFF WHERE DID=1;

SUM

The sum aggregate function computes the sum of all the tuple values of the column. It works on the numeric fields only.

Syntax:

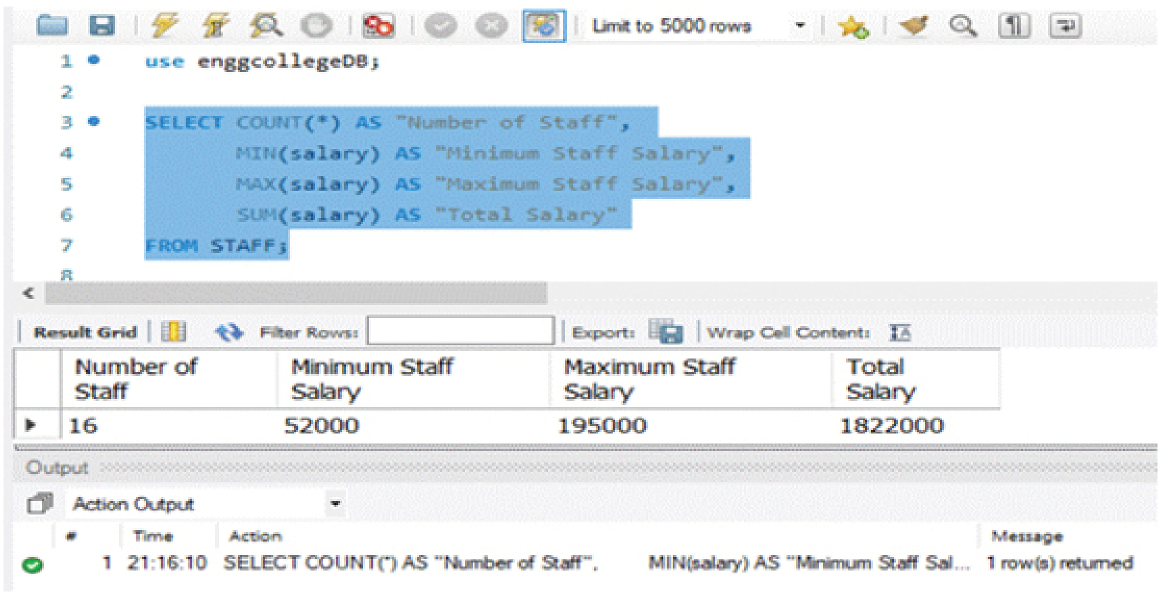
`SUM()` or `SUM( [ALL|DISTINCT] expression )`

Example:

SELECT SUM(salary) FROM STAFF;

SELECT did, SUM(salary) AS DEPT_Salary FROM STAFF GROUP BY did;

Figure 4.23 shows the execution of the Aggregate functions:



```
use enggcollegeDB;

SELECT COUNT(*) AS "Number of Staff",
       MIN(salary) AS "Minimum Staff Salary",
       MAX(salary) AS "Maximum Staff Salary",
       SUM(salary) AS "Total Salary"
FROM STAFF;
```

Number of Staff	Minimum Staff Salary	Maximum Staff Salary	Total Salary
16	52000	195000	1822000

```
#  Time      Action                                                                   Message
1  21:16:10  SELECT COUNT(*) AS "Number of Staff",    MIN(salary) AS "Minimum Staff Sal...   1 row(s) returned
```

Figure 4.23: *Aggregate Function - Count, MIN, MAX and SUM*

The GROUP BY and HAVING Clause

The aggregate functions will return the result based on the tuples which satisfies the selection criteria. But sometimes, it is required to group them based on the similarity instead of combining into a single value. So, when we want to apply the aggregate operations based on the similarity to each individual tuple of the table, we specify it in SQL using the group by clause.

In short, the Group by clause is used to group the rows that have the same value using the attribute/attributes listed in the group by clause (Group By is used to summarize or aggregate the data series).

Syntax:

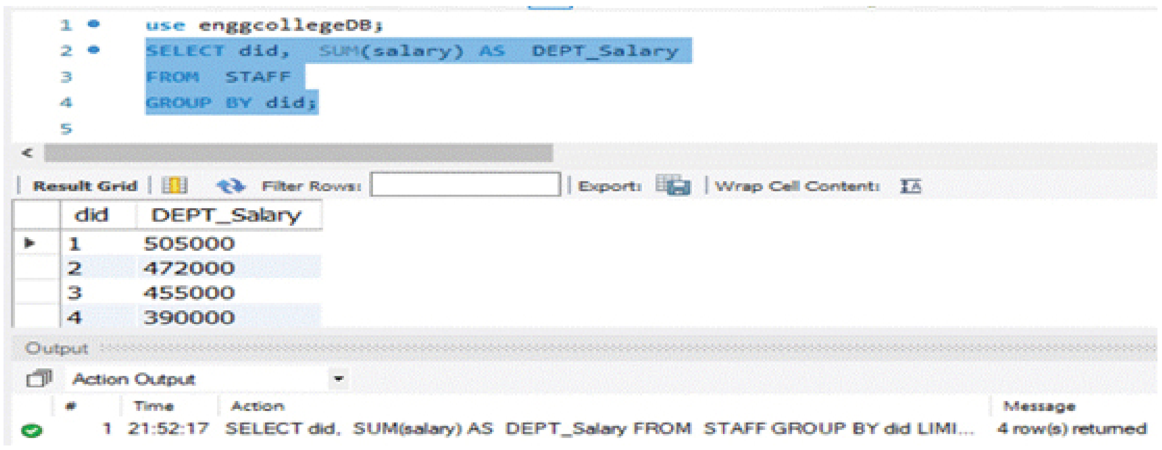
```
SELECT expression1, expression2, ... expression_n,
        aggregate_function (aggregate_expression)
FROM tables
[WHERE conditions]
GROUP BY expression1, expression2, ... expression_n
[ORDER BY expression [ ASC | DESC ]];
```

Example: *Figure 4.24* shows the Group By clause to compute the department wise salary of the staff:

SELECT did, SUM(salary) AS DEPT_Salary

FROM STAFF

GROUP BY did;



```
use enggcollegeDB;
SELECT did,  SUM(salary) AS  DEPT_Salary
FROM   STAFF
GROUP  BY  did;
```

did	DEPT_Salary
1	505000
2	472000
3	455000
4	390000

```
#   Time      Action                                                            Message
1   21:52:17  SELECT did,  SUM(salary) AS  DEPT_Salary FROM  STAFF GROUP BY did LIMI...   4 row(s) returned
```

Figure 4.24: *Group By clause*

Group By with HAVING clause

The Group by clause combines the set of tuples based on columns. The Having clause is used to impose the condition on the group by clause to retrieve values if we want to set a filter on this column. Since the `WHERE` clause filters the individual tuples and can't be used to filter the set of tuples or groups, SQL provides a Having clause. SQL allows to sort the grouped records using the `ORDER BY` clause as well.

The simplified syntax of the having clause is as follows:

SELECT attribute list, aggregate_function (aggregate_expression)

FROM tables

[WHERE conditions]

GROUP BY [grouping-list expression1, expression2, ... expression_n]

HAVING condition

ORDER BY Column_name [ASC | DESC] ;

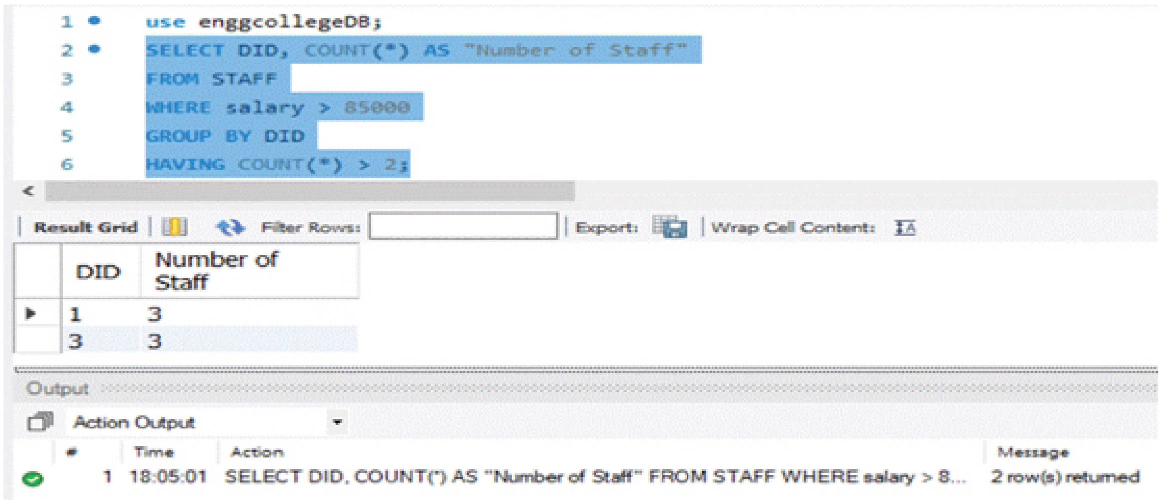
Example: *Figure 4.25* shows the Group By with the Having clause to filter the content of the Group By clause:

SELECT DID, COUNT(*) AS "Number of Staff"

FROM STAFF

WHERE salary > 85000

GROUP BY DID
HAVING COUNT(*) > 2;



```
use enggcollegeDB;
SELECT DID, COUNT(*) AS "Number of Staff"
FROM STAFF
WHERE salary > 85000
GROUP BY DID
HAVING COUNT(*) > 2;
```

DID	Number of Staff
1	3
3	3

```
#   Time      Action                                                                 Message
1   18:05:01  SELECT DID, COUNT(*) AS "Number of Staff" FROM STAFF WHERE salary > 8...  2 row(s) returned
```

Figure 4.25: *Group By with HAVING clause*

This query will compute the number of staff members as the query is COUNT(*), but later it will set the filter based on the where clause and list only those tuples which have a salary greater than 85000. Then, it will be grouped based on the department id and the last condition is that the number of tuples must be larger than two in the group which will then display the department in the query outcome, otherwise it will not. This indicates that the where and the having clause can be used in the same query.

The users can run the query in parts to check the working of the query with appropriate comment lines. For example, just comment the Having clause and check the result, then comment the where clause and check the outcome. In short, if we mask the effect of the filter, the outcome of the query changes as per the filter applied.

Some observations about the `HAVING` clause are as follows:

- Where clause filters the individual tuples, whereas the `HAVING` clause operates on the result of the group by clause or group of records.
- It will return the groups which satisfy the criteria set by the `HAVING` clause.
- To apply the `HAVING` clause, `GROUP BY` is required.
- `WHERE` and `HAVING` can be used in the same query.

Views

Views are virtual tables. The SQL views an be described as virtual tables, or it is just an unmaterialized relation. By unmaterialized, we store a definition rather than the actual data. The views are built on top of the other tables and do not hold the actual data. Views require very little space as it stores only the definitions of the view. A view has rows and columns as that of a real table in the database. A user can create a view based on the requirement, i.e., a View can either have all the rows of a table or specific rows based on certain requirements. It may combine the data from multiple sources and avoid the creation of separate table/tables. A good thing about views is that it can be created using tables of the same database or different databases.

One of the purposes of the view is to provide a security mechanism, as it provides advantages over the tables. It represents a subset of data from tables or more than one database. Whenever the data in any table, which is part of view, is changed, the effect is reflected in the view. The view can be classified as a simple view and complex view. The simple view is based on a single table, whereas the complex view is based on multiple tables and may contain join, group by, and order by clauses. A View has many advantages and security is one of them. It degrades the performance as compared to that of a real table, as it is created on top of the real tables. The query processor must translate the queries from view to queries to the underlying source tables.

A view is defined in SQL using the create view command. To create a view, we must give a view name and the required query expression which computes the view. The simplified syntax of the create view command is as follows:

```
CREATE
    [OR REPLACE]
        VIEW view_name [(column_list)]
        AS select_statement;
```

Specify the name of the view that you want to create after the `CREATE VIEW` command. The name of the view must be unique in the database as it shares the same namespace. As views are created with a purpose, the name of columns can be listed explicitly.

The following is an example:

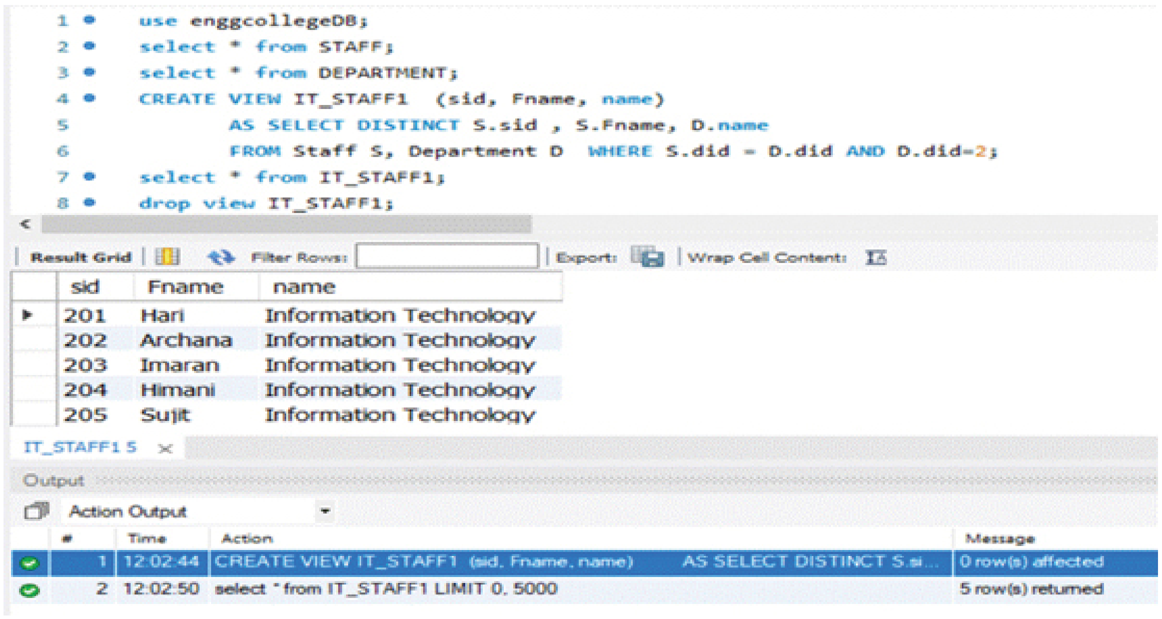
```
CREATE VIEW IT_STAFF1 (Sid, Fname, Name)
    AS SELECT DISTINCT S.sid , S.Fname, D.name
    FROM Staff S, Department D WHERE S.did = D.did AND D.did=2;
```

The following is the MySQL Code:

```
use enggcollegeDB;
select * from STAFF;
select * from DEPARTMENT;
CREATE VIEW IT_STAFF1 (sid, Fname, name)
    AS SELECT DISTINCT S.sid , S.Fname, D.name
    FROM Staff S, Department D WHERE S.did = D.did AND D.did=2;
select * from IT_STAFF1;
drop view IT_STAFF1;
```

A few more points related to the view are:

- Views may be used to provide the required information or a summary, while keeping the other details hidden from the end users from the underlying relation(s).
- Views can be treated as if they were materialized relations.
- The system translates a `SELECT` on a view into `SELECTS` on the materialized relations.
- In MySQL, the maximum number of tables that can be referenced in the definition of a view is 61. If the view is a complex view, then the update anomaly becomes more difficult to handle. As a result, many SQL-based database systems impose the constraints on the modifications allowed through the views.
- To drop the view, the `DROP VIEW` command is used.
- *Figure 4.26* shows the execution of views in MySQL.



```
use enggcollegeDB;
select * from STAFF;
select * from DEPARTMENT;
CREATE VIEW IT_STAFF1  (sid, Fname, name)
        AS SELECT DISTINCT S.sid , S.Fname, D.name
        FROM Staff S, Department D  WHERE S.did = D.did AND D.did=2;
select * from IT_STAFF1;
drop view IT_STAFF1;
```

sid	Fname	name
201	Hari	Information Technology
202	Archana	Information Technology
203	Imaran	Information Technology
204	Himani	Information Technology
205	Sujit	Information Technology

```
#   Time      Action                                                                   Message
1   12:02:44  CREATE VIEW IT_STAFF1 (sid, Fname, name)     AS SELECT DISTINCT S.si...  0 row(s) affected
2   12:02:50  select * from IT_STAFF1 LIMIT 0, 5000                                    5 row(s) returned
```

Figure 4.26: *Views in SQL*

Joins

Definition: Combining the columns from one or more relations which satisfy the join condition.

Types of JOINS: JOINS are majorly divided into two categories, which are as follows:

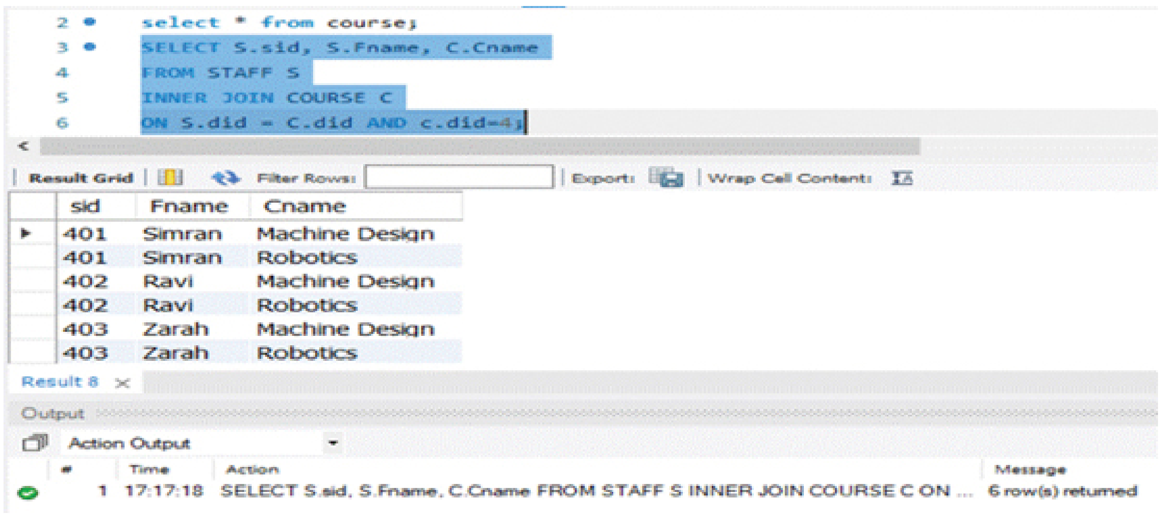
I Inner Joins:

- Natural join
- EQUI join
- Theta join

II Outer join:

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

Inner Joins: Select the rows that have matching values from the tables that are being joined, as shown in *Figure 4.27*:



```
select * from course;
SELECT S.sid, S.Fname, C.Cname
FROM STAFF S
INNER JOIN COURSE C
ON S.did = C.did AND c.did=4;
```

```
sid   Fname    Cname
401   Simran   Machine Design
401   Simran   Robotics
402   Ravi     Machine Design
402   Ravi     Robotics
403   Zarah    Machine Design
403   Zarah    Robotics
```

```
#   Time       Action                                                              Message
1   17:17:18   SELECT S.sid, S.Fname, C.Cname FROM STAFF S INNER JOIN COURSE C ON ...   6 row(s) returned
```

Figure 4.27: *Inner Join*

Natural join: It is applied on all the columns in the two tables that have the same column name and also the same data type. It returns all the rows from both the tables that have equal values in all the matched columns. Remember, if the name of the columns are the same in the two tables but if the data type is mismatched, an error will appear.

Syntax:

SELECT *

FROM table1

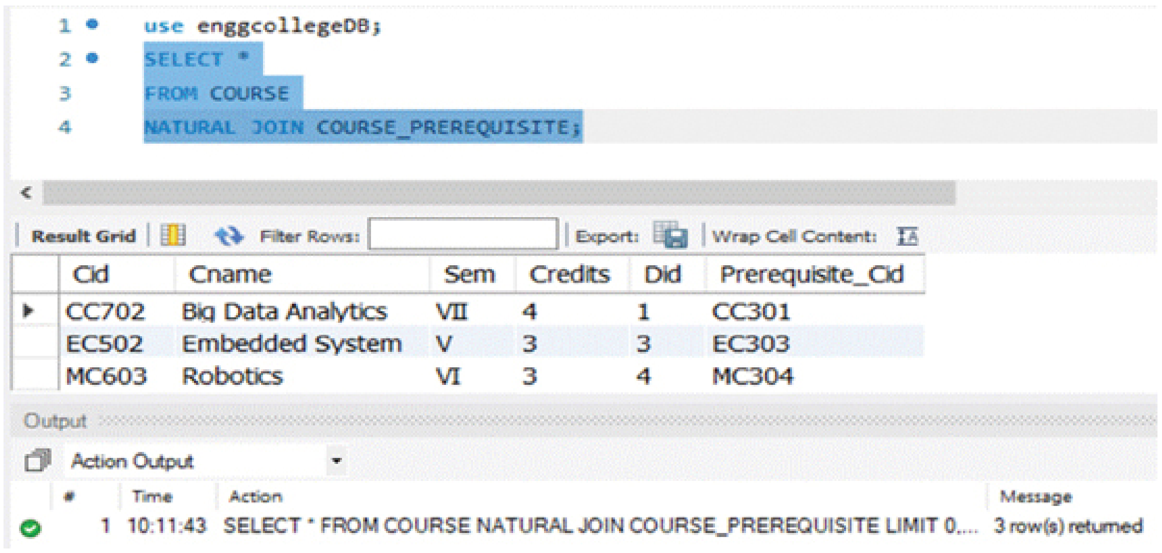
NATURAL JOIN table2;

Example: Display all the courses which have prerequisite courses. It shows the Natural join on the `COURSE` and `COURSE_PREREQUISITE` relations, as shown in *Figure 4.28*:

SELECT *

FROM COURSE

NATURAL JOIN COURSE_PREREQUISITE;



```
use enggcollegeDB;
SELECT *
FROM COURSE
NATURAL JOIN COURSE_PREREQUISITE;
```

	Cid	Cname	Sem	Credits	Did	Prerequisite_Cid
▶	CC702	Big Data Analytics	VII	4	1	CC301
	EC502	Embedded System	V	3	3	EC303
	MC603	Robotics	VI	3	4	MC304

```
#   Time      Action                                                             Message
1   10:11:43  SELECT * FROM COURSE NATURAL JOIN COURSE_PREREQUISITE LIMIT 0,...  3 row(s) returned
```

Figure 4.28: *Natural Join*

EQUI join

Equijoin is a special type of natural join of two relations, R and S, over a specified attribute. The attributes must be with the same domain. It holds only the equality conditions between a pair of attributes. The resultant relation carries those tuples whose values of the two attributes will be equal and only one attribute will appear in the result.

Syntax:

SELECT column_list

FROM table1, table2....

WHERE table1.column_name = table2.column_name ;

or

Syntax:

SELECT column_list

FROM table1

JOIN table2

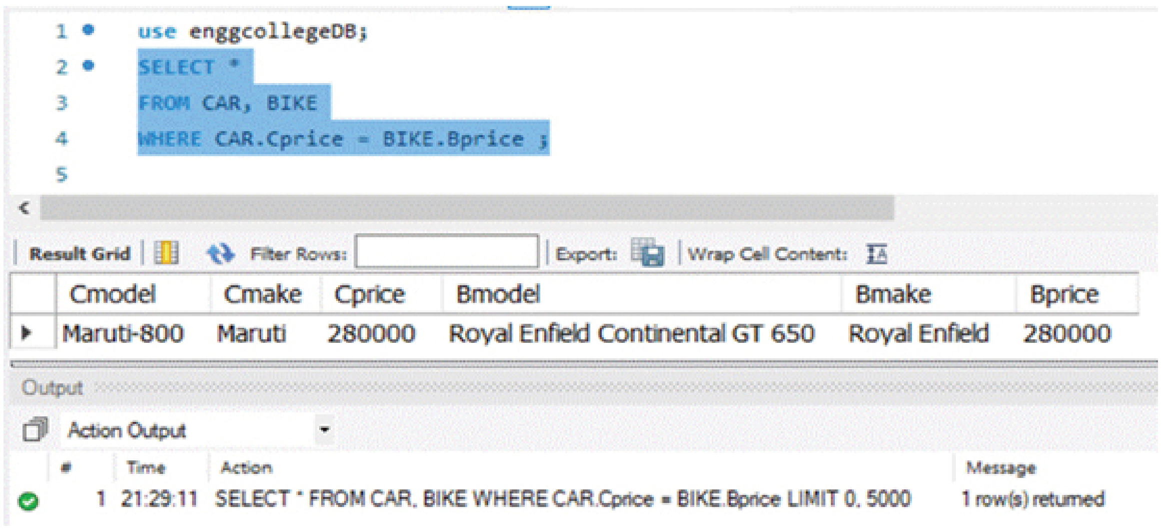
[ON (join_condition)] ;

Example: *Figure 4.29* shows the execution of EquiJoin:

SELECT *

FROM CAR, BIKE

WHERE CAR.Cprice = BIKE.Bprice ;



```
use enggcollegeDB;
SELECT *
FROM CAR, BIKE
WHERE CAR.Cprice = BIKE.Bprice ;
```

Cmodel	Cmake	Cprice	Bmodel	Bmake	Bprice
Maruti-800	Maruti	280000	Royal Enfield Continental GT 650	Royal Enfield	280000

```
#  Time      Action                                                            Message
1  21:29:11  SELECT * FROM CAR, BIKE WHERE CAR.Cprice = BIKE.Bprice LIMIT 0, 5000  1 row(s) returned
```

Figure 4.29: *Equi Join*

Theta Join

The most general form of inner join is Theta join. This is the general join which everyone wants to use as it allows to specify the condition while performing the join operation. Theta join produces the tuples from both the relation operands that satisfy the join condition. Theta join can use any conditions in the selection criteria.

Syntax:

SELECT *

FROM table1, table2

WHERE table1.column1 operator table2.column2 ;

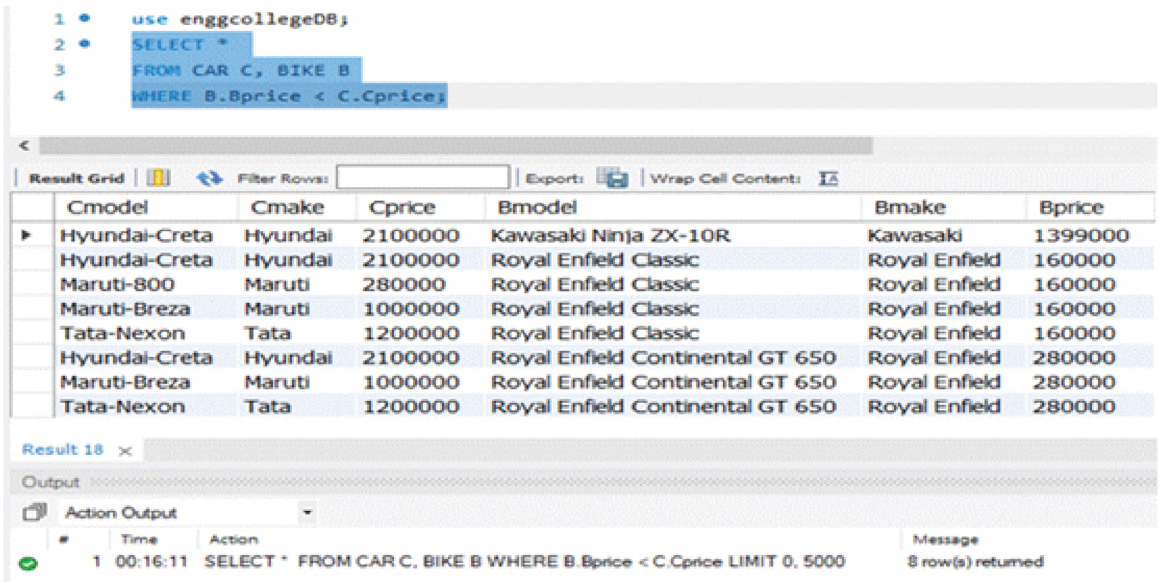
In the preceding syntax, the operator could be any operator other than the equal operator.

Example: Theta Join of CAR and BIKE relations, as shown in *Figure 4.30*:

SELECT *

FROM BIKE B, CAR C

WHERE B.Bprice < C.Cprice;



```
use enggcollegeDB;
SELECT *
FROM CAR C, BIKE B
WHERE B.Bprice < C.Cprice;
```

Cmodel	Cmake	Cprice	Bmodel	Bmake	Bprice
Hyundai-Creta	Hyundai	2100000	Kawasaki Ninja ZX-10R	Kawasaki	1399000
Hyundai-Creta	Hyundai	2100000	Royal Enfield Classic	Royal Enfield	160000
Maruti-800	Maruti	280000	Royal Enfield Classic	Royal Enfield	160000
Maruti-Breza	Maruti	1000000	Royal Enfield Classic	Royal Enfield	160000
Tata-Nexon	Tata	1200000	Royal Enfield Classic	Royal Enfield	160000
Hyundai-Creta	Hyundai	2100000	Royal Enfield Continental GT 650	Royal Enfield	280000
Maruti-Breza	Maruti	1000000	Royal Enfield Continental GT 650	Royal Enfield	280000
Tata-Nexon	Tata	1200000	Royal Enfield Continental GT 650	Royal Enfield	280000

```
#   Time       Action                                                           Message
1   00:16:11   SELECT * FROM CAR C, BIKE B WHERE B.Bprice < C.Cprice LIMIT 0, 5000   8 row(s) returned
```

Figure 4.30: *Theta Join*

Outer joins: In SQL, whenever the Join operations are carried out, the default assumption is Inner Join. Inner join always takes place on the join condition specified on the columns of the participating relations. It will return only those tuples which satisfy the join condition. But, if someone is expecting all tuples from the participating relations, then the Outer Join can be used. Based on the requirements, the outer join can be classified as – Left outer join, right outer join, and Full outer join.

A. Left Outer Join

All the tuples from the left relation (say R) are included in the resulting relation. If there are tuples in R without any matching tuple in the right relation S, then the S-attributes of the resulting relation are supposed to be set as NULL.

Syntax:

SELECT column_name(s)

FROM table1

LEFT JOIN table2

ON table1.column_name = table2.column_name;

Example: *Figure 4.31* shows the execution of the Left outer join:

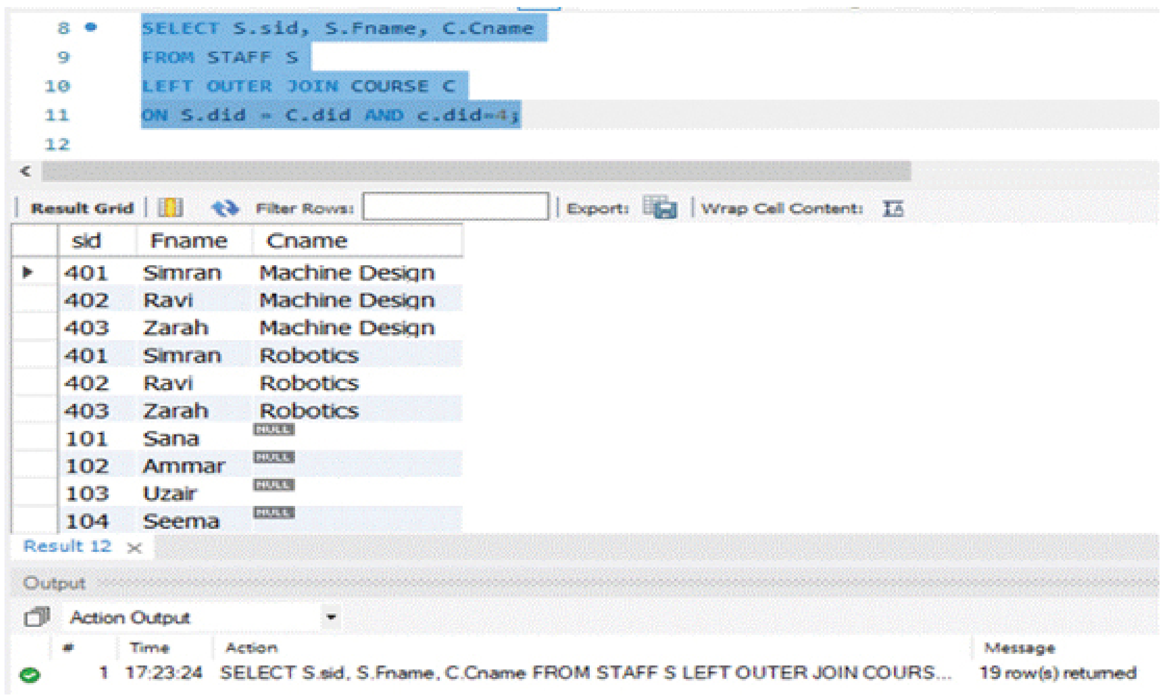
SELECT S.sid, S.Fname, C.Cname

FROM STAFF S

LEFT OUTER JOIN COURSE C

ON S.did = C.did AND c.did=4;

It will find the match based on the Department id and list, as shown in *Figure 4.31*:



```
SELECT S.sid, S.Fname, C.Cname
FROM STAFF S
LEFT OUTER JOIN COURSE C
ON S.did = C.did AND c.did=4;
```

sid	Fname	Cname
401	Simran	Machine Design
402	Ravi	Machine Design
403	Zarah	Machine Design
401	Simran	Robotics
402	Ravi	Robotics
403	Zarah	Robotics
101	Sana	NULL
102	Ammar	NULL
103	Uzair	NULL
104	Seema	NULL

```
#  Time      Action                                                          Message
1  17:23:24  SELECT S.sid, S.Fname, C.Cname FROM STAFF S LEFT OUTER JOIN COURS...  19 row(s) returned
```

Figure 4.31: *Left Outer Join*

B. Right Outer Join

All the tuples from the right relation (say S) are included in the resulting relation. If there are tuples in S without any matching tuple in the Left relation R, then the R-attributes of the resulting relation are supposed to be set as NULL.

Syntax:

SELECT column_name(s)

FROM table1

RIGHT JOIN table2

ON table1.column_name = table2.column_name;

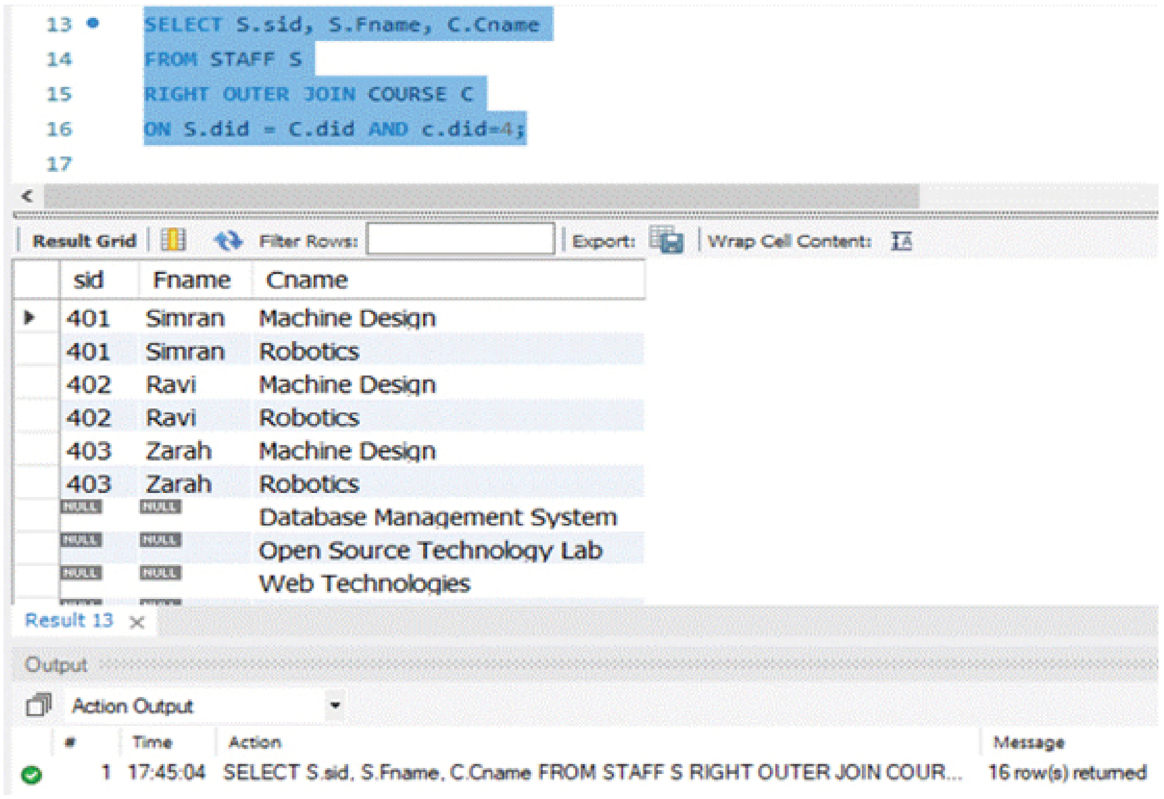
Example: *Figure 4.32* shows the execution of the Right outer join:

SELECT S.sid, S.Fname, C.Cname

FROM STAFF S

RIGHT OUTER JOIN COURSE C

ON S.did = C.did AND c.did=4;



```
SELECT S.sid, S.Fname, C.Cname
FROM STAFF S
RIGHT OUTER JOIN COURSE C
ON S.did = C.did AND c.did=4;
```

```
sid   Fname   Cname
401   Simran  Machine Design
401   Simran  Robotics
402   Ravi    Machine Design
402   Ravi    Robotics
403   Zarah   Machine Design
403   Zarah   Robotics
NULL  NULL    Database Management System
NULL  NULL    Open Source Technology Lab
NULL  NULL    Web Technologies
```

```
#  Time      Action                                                      Message
1  17:45:04  SELECT S.sid, S.Fname, C.Cname FROM STAFF S RIGHT OUTER JOIN COUR...  16 row(s) returned
```

Figure 4.32: *Right Outer Join*

C. Full Outer Join

All the tuples from both the participating relations are included in the resulting relation. If there are no matching tuples for both the relations, their respective unmatched attributes are supposed to be set as NULL.

Syntax:

SELECT column_name(s)

FROM table1

FULL OUTER JOIN table2

ON table1.column_name = table2.column_name;

Similarly, based on the queries given for the left outer join and right outer join, the users can write the queries related to the Full outer join using the preceding syntax.

Nested and Complex queries

In MySQL, the subquery can be used to write complex queries.

A subquery can be nested inside another SQL query. Here, an inner query is known as a subquery, while the query which holds the subquery is known as an outer query.

The inner query executes first, and it returns the row/rows as an input to the outer query.

A subquery may occur in a `SELECT` clause, `FROM` clause, or a `WHERE` clause. It must be enclosed within the parentheses.

Syntax:

```
Select column_list
from table1
WHERE expression operator
            (SELECT column_list from table2);
```

You can use the comparison operators, such as >, <, or =, for single row queries, and ANY, SOME, or ALL comparison operators can be used for multiple-row queries.

MySQL subquery with =, >, < operators

For example, to display the details of staff who have a maximum salary, the following query can be fired:

```
SELECT Sid, Fname, Designation, Salary, Email
FROM STAFF
WHERE
    Salary = (SELECT MAX(Salary) FROM STAFF);
```

Other than the equal operator (=), we can also make use of the other comparison operators, such as greater than (>) and less than(<), based on the requirements of the users.

For example, you can find the staff whose salary is greater than the average salary using a subquery, as shown as follows:

```
SELECT Sid, Fname, Designation, Salary, Email
```

```
FROM STAFF
WHERE Salary > (SELECT AVG(Salary) FROM STAFF);
```

In the preceding query, firstly a subquery will get executed which calculates the average salary from STAFF using the AVG aggregate function. The result of the inner query will become the input to the outer query. Then, the outer query will execute and return the staff whose salary is greater than the average salary.

MySQL subquery with IN and NOT IN operators

If a subquery result returns more than one value, we cannot use the single row operators. We can use multi-row operators, such as the `IN` or `NOT IN` operators, in the `WHERE` clause.

Example: Display the first name and Did of all the staff who are in Computer Engineering or Information Technology departments; the query to this is as follows:

```
SELECT fname, Did
FROM STAFF
WHERE Did IN
    (SELECT Did from DEPARTMENT
    WHERE Dname== "Computer Engineering" or Dname=" Information
    Technology");
```

Example: Display all the course's id and name which does not belong to the department of Computer Engineering or Information Technology departments, as shown as follows:

```
SELECT Cid, Cname
FROM COURSE
WHERE Did NOT IN
    (SELECT Did from DEPARTMENT
    WHERE Dname="Computer Engineering" or Dname="Information
    Technology");
```

Example: Display the first name, Designation, and Salary of staff whose salary is less than the salary of all the staff whose Designation is Professor or Associate Professor, as shown as follows:

```
SELECT fname, Designation, Salary
```

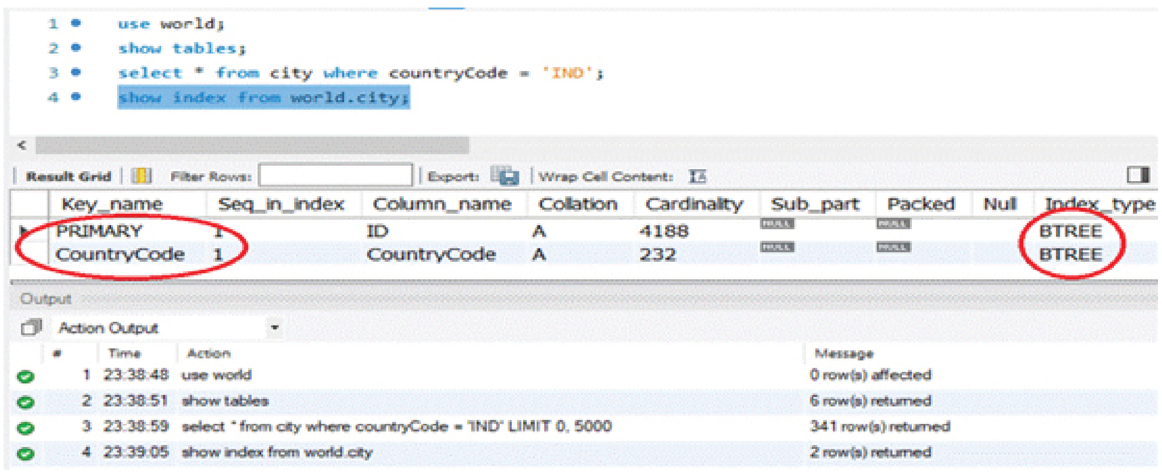
```
FROM STAFF
WHERE Salary < ALL
    (SELECT Salary from STAFF
    WHERE Designation = “Professor” or Designation = “Associate
    Professor”);
```

Indexing in SQL

To understand the concept and importance of indexing, let us consider a simple example. Suppose you are interested in a topic of databases. How will you search it immediately? There are two different ways – either look at the contents of the book or the index of the book. The index of the book, most probably the few last pages, gives the index of the keywords in which the user is interested and can directly pin-point to the page in which he/she can find. If it appears on more than one page, then all those pages are listed to get the details of the topic of interest. The keywords are in the alphabetical order, which makes it really easy for us to scan the index. This is the most efficient way of searching the contents from a book. Similarly, the index in SQL gives an edge to access the data from the table. An index is a special lookup table, or an orderly arrangement used to logically access the records from a table with the specific column values quickly.

A database index is a data structure that improves the speed of data retrieval on a table. Indexes can be created using one or more columns for both rapid random lookups and efficient ordering of access to records. Index uses the B-tree data structure to find the records associated with the key values quickly and efficiently, but it is essential that the correct indexes are defined for each table. The impact of the index won't be noticed in a small database, but it becomes noticeable once the table size grows. The query optimizer may use the indexes to quickly locate the data without having to scan every row in a table for a given query.

Indexes are also a type of tables which keep the primary key or index field and a pointer to each record into the actual table. By default, MySQL uses `BTREE` for indexing. If a table has indexes, applying the `SELECT` command would be fast on the tables as compared to the `INSERT` and `UPDATE` commands which take more time on the tables, because while performing the insert or update operations, a database also needs to insert or update the index values. Indexes can be displayed in MySQL by using the **`SHOW INDEX`** command, as shown in *Figure 4.33*:



```
use world;
show tables;
select * from city where countryCode = 'IND';
show index from world.city;
```

Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
PRIMARY	1	ID	A	4188	NULL	NULL		BTREE
CountryCode	1	CountryCode	A	232	NULL	NULL		BTREE

```
#  Time      Action                                                      Message
1  23:38:48  use world                                                   0 row(s) affected
2  23:38:51  show tables                                                 6 row(s) returned
3  23:38:59  select * from city where countryCode = 'IND' LIMIT 0, 5000  341 row(s) returned
4  23:39:05  show index from world.city                                  2 row(s) returned
```

Figure 4.33: *Index in MySQL*

The users can create the indexes by using the `CREATE INDEX` or `CREATE UNIQUE INDEX` commands, as shown in *Figure 4.34*.

Syntax: If you want to `CREATE INDEX` when the duplicate values are allowed, fire the following query:

CREATE INDEX index_name

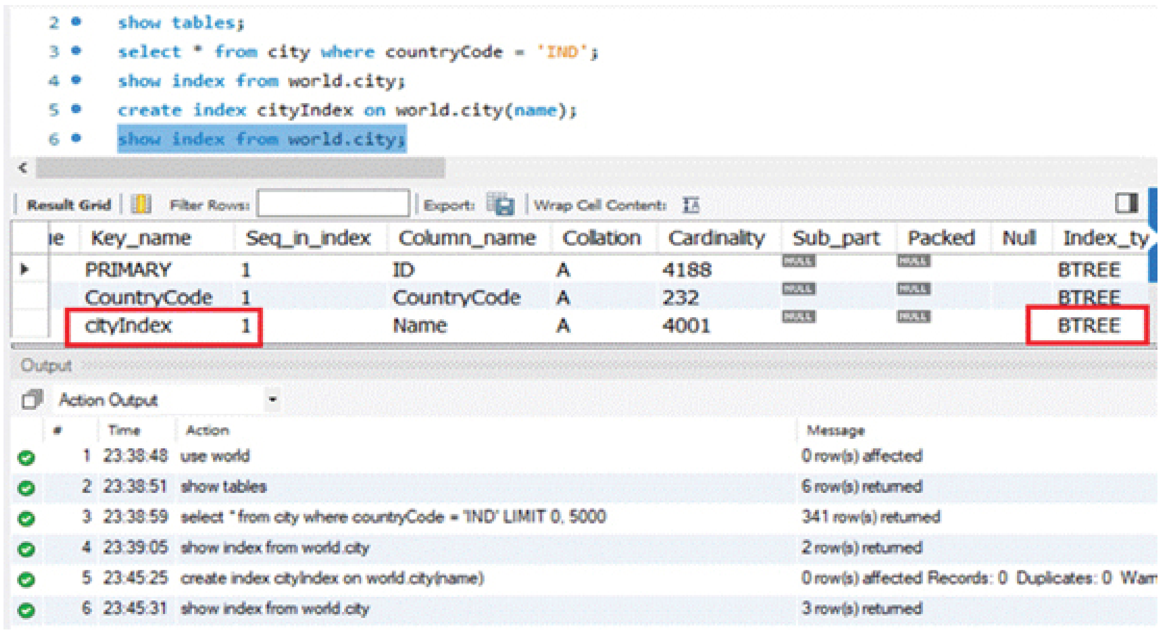
ON table_name (column1, column2, ...);

Syntax: If you want to `CREATE INDEX` when duplicate values are not allowed, fire the following query:

CREATE UNIQUE INDEX index_name

ON table_name (column1, column2, ...);

Refer to Figure 4.34 that illustrates creating a new index in MySQL:



```
show tables;
select * from city where countryCode = 'IND';
show index from world.city;
create index cityIndex on world.city(name);
show index from world.city;
```

ie	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_ty
	PRIMARY	1	ID	A	4188	NULL	NULL		BTREE
	CountryCode	1	CountryCode	A	232	NULL	NULL		BTREE
	cityIndex	1	Name	A	4001	NULL	NULL		BTREE

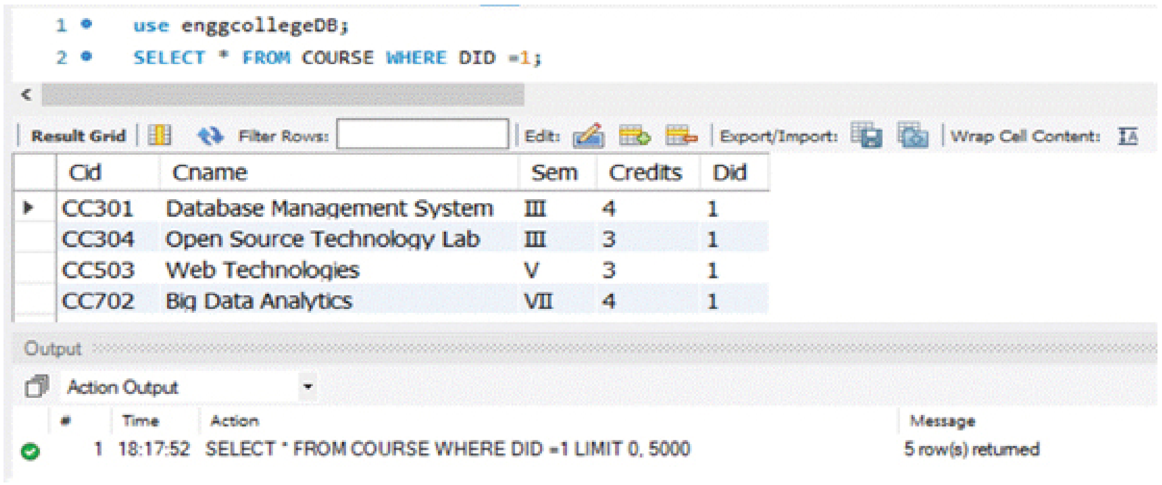
```
#  Time      Action                                                      Message
1  23:38:48  use world                                                   0 row(s) affected
2  23:38:51  show tables                                                 6 row(s) returned
3  23:38:59  select * from city where countryCode = 'IND' LIMIT 0, 5000  341 row(s) returned
4  23:39:05  show index from world.city                                  2 row(s) returned
5  23:45:25  create index cityIndex on world.city(name)                  0 row(s) affected Records: 0  Duplicates: 0  Warn
6  23:45:31  show index from world.city                                  3 row(s) returned
```

Figure 4.34: *Creating new Index in MySQL*

As shown in *Figure 4.34*, the users can create indexes on a particular column using the Create Index command, and MySQL, by default, creates a B-tree index.

Cursor

Relational database operates on a set of rows. For example, when we fire a query using the `SELECT` command, it returns a set of rows, as shown in *Figure 4.35*:



```
use enggcollegeDB;
SELECT * FROM COURSE WHERE DID =1;
```

Cid	Cname	Sem	Credits	Did
CC301	Database Management System	III	4	1
CC304	Open Source Technology Lab	III	3	1
CC503	Web Technologies	V	3	1
CC702	Big Data Analytics	VII	4	1

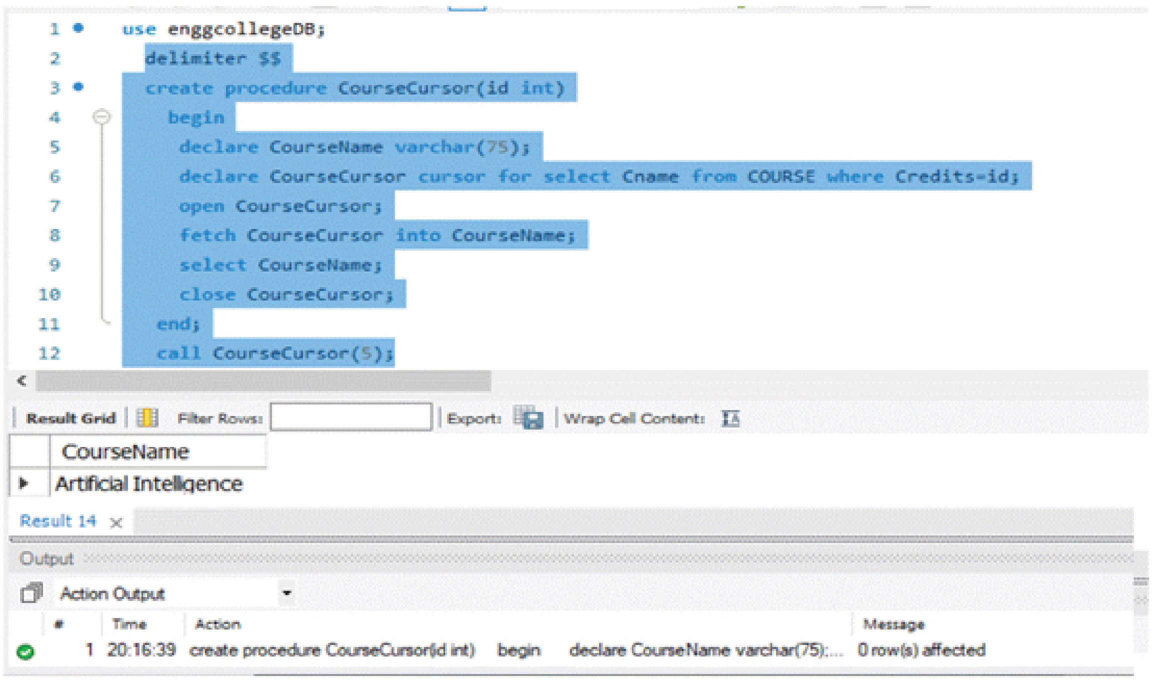
```
#  Time      Action                                           Message
1  18:17:52  SELECT * FROM COURSE WHERE DID =1 LIMIT 0, 5000  5 row(s) returned
```

Figure 4.35: *SELECT in SQL*

If some application requires to process individual rows, in the programming languages, you will probably use a loop, like `FOR` or `WHILE`, to iterate through a single row at a time and the database programmers can use the cursors to achieve the same result. A cursor is a temporary work area created in the system memory which enables sequential processing of the rows in a result set. The set of rows a cursor holds is called the active set with a pointer that identifies a current row. In short, a cursor is used as a control to go through the database records, and it is similar to the control loops of the programming languages.

Example: CourseCursor using MySQL, as shown in *Figure 4.36*:

```
use enggcollegeDB;
 delimiter $$
 create procedure CourseCursor(id int)
     begin
     declare CourseName varchar(75);
     declare CourseCursor cursor for select Cname from COURSE where
     Credits=id;
     open CourseCursor;
     fetch CourseCursor into CourseName;
     select CourseName;
     close CourseCursor;
 end;
 call CourseCursor(5);
```



```
use enggcollegeDB;
  delimiter $$
  create procedure CourseCursor(id int)
    begin
     declare CourseName varchar(75);
     declare CourseCursor cursor for select Cname from COURSE where Credits=id;
     open CourseCursor;
     fetch CourseCursor into CourseName;
     select CourseName;
     close CourseCursor;
   end;
   call CourseCursor(5);
```

```
CourseName
Artificial Intelligence
```

```
#  Time      Action                                                                     Message
1  20:16:39  create procedure CourseCursor(id int)  begin  declare CourseName varchar(75);...  0 row(s) affected
```

Figure 4.36: *Cursor in MySQL*

Figure 4.36 demonstrates the Cursor example with the input parameters which accept the input values. A cursor needs to be declared and assigned a name, before it can be used. Open, fetch, and close are used to open the cursor, fetch the value, and remove the cursor. Closed cursors can be opened again. The call cursor will execute the CourseCursor and return the records, that is, it will call a cursor with required credits and it will return the Course name as Artificial Intelligence.

Triggers

A trigger is a stored program which is invoked automatically when any event occurs in the database, such as inserting data into table or updating or deleting some data from the table. Triggers can also be used to check the integrity of the data.

For example, suppose, we have to always ensure that the salary of staff should be less than or equal to the academic coordinator's salary, if it is not null (suppose this is the constraint for Engineering college database). Here, we want to check the salary of a staff member with the salary of the academic coordinator; if it's less, then it will make a change into the database, else reject the record to insert/update into the database. So, to fulfill this requirement, we can make use of triggers. Write a trigger to ensure that the college Academic coordinator must always have a higher or equal salary than any staff that he or she monitored. Users can create Trigger for the same, based on the following example.

We can define a trigger that is invoked automatically before a new record is inserted into a table.

Syntax:

```
CREATE TRIGGER trigger_name trigger_time trigger_event
ON table_name
FOR EACH ROW
BEGIN
...
END;
```

In the preceding syntax, `trigger_name` is the name of the trigger, `trigger_time` is the time of trigger activation which can be before or after, `trigger_event` can be `INSERT`, `UPDATE`, or `DELETE`. `table_name` is the name of the table, which is associated with triggers. `BEGIN…END` is the block in which we need to define the logic for the trigger.

Example:

```
Create Trigger check_salary BEFORE INSERT
ON STAFF
FOR EACH ROW

BEGIN
IF NEW.Salary < 0 THEN SET NEW.Salary = 1000 ;
END
```

The preceding query will create a trigger named `check_salary` and it will be invoked before inserting any row into the `STAFF` table. As `Salary` cannot be negative, triggers will get invoked if the user is trying to insert a tuple with the salary value less than 0 and automatically assign the default salary value 1000 in the table.

Embedded SQL

SQL is a declarative language – it is nonprocedural. It is not a programming language as it does not have loops to perform repetitive tasks or control the structures to make the decisions. In SQL, the focus is on the actual result and not on how to obtain it. The embedded SQL enables the integration of SQL with a general-purpose programming language like Java or Python. Embedded SQL takes advantage of the strengths of host languages and SQL. The SQL statements could be used wherever a statement in the host language is permitted, and it is the predominant way to integrate the SQL statements with the host language. The host language can be used to perform the tasks, such as user interaction, report printing, and providing GUI, and sending the result to GUI.

While margining the two languages or taking the advantages of both the environment, the main problem is of impedance mismatch – the data model of SQL is significantly different than that of the Java/Python. Therefore, passing the data between the host language and SQL is not straightforward. For Java language, it is a Java Database Connectivity (JDBC) library, an advancement over ODBC which is platform dependent. JDBC API acts as an interface to establish a link between the Java and Database. The steps for connecting a Java application with Database using JDBC are as follows:

1. Load/register the driver
2. Create a connection
3. Create a statement
4. Execute query
5. Close the connection

First, import the complete SQL package with the import statement "`import java.sql.*`" or the required classes as `SQLException`, `DriverManager`, and `Connection` from the `java.sql.*` package.

1. **Load/Register the driver**

   To open a communication channel with the database, it requires the initialization of the driver. It can be achieved through the `forName()` method of `Class`. This method is used to register and dynamically load the driver class. The syntax of the `forName()` method is as follows:

   public static void forName(String className) throws
   ClassNotFoundException

2. Create a connection

After the registration of the driver, we can build up the associations using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with the database.

The `getConnection()` method of the `DriverManager` class is used to establish the connection with the database.

The syntax of the `getConnection()` method is as follows:

public static Connection getConnection(String url) throws SQLException

public static Connection getConnection(String url, String user_name, String password) throws SQLException

Connection con = DriverManager.getConnection(url, user, password)

User: The username, i.e., the database user that will be used to connect to MySQL.

Password: The password of the database user

Con: Is a reference to the Connection interface.

URL: Uniform Resource Locator.

For MySQL, you use the `jdbc:mysql://localhost:3306/enggcollegeDB`, i.e., you are connecting to the MySQL with the server-name localhost, port 3306, and database enggcollegeDB.

When you work with MySQL, if anything goes wrong, for example, the database server is not available or username and password is wrong, then JDBC throws a SQLException. Therefore, when you create a Connection object, you should always put it inside a try-catch-block, as shown in *Figure 4.37*.

3. Create a Statement

The `createStatement()` method of the Connection interface is used for building and submitting an SQL statement to the database. The object of statement is responsible to execute the queries with the database.

The syntax of the `createStatement()` method is as follows:

```
public Statement createStatement()throws SQLException
```

Example: `Statement stmt=con.createStatement();`

4. Execute query

The `executeQuery()` method of the Statement is used to execute the queries to the database from the Java program. This method returns the object of `ResultSet` and can be used to get the records of a table.

The syntax of the `executeQuery()` method is as follows:

public ResultSet executeQuery(String sql)throws SQLException

Use the appropriate `Resultset get` methods to retrieve the data from the result set. Refer to the following example:

```
    String Course = rs.getString("Cname");
  int Credit = rs.getInt("Credits");
```

5. Close the connection

The `close()` method of the Connection interface is used to close the connection. It is a good practice to use the close method to close the connections instead of relying on the JVM's garbage collection as it requires explicitly closing all the database resources. By closing the connection, object statement and ResultSet will be closed automatically.

The syntax of the `close()` method is as follows:

public void close()throws SQLException

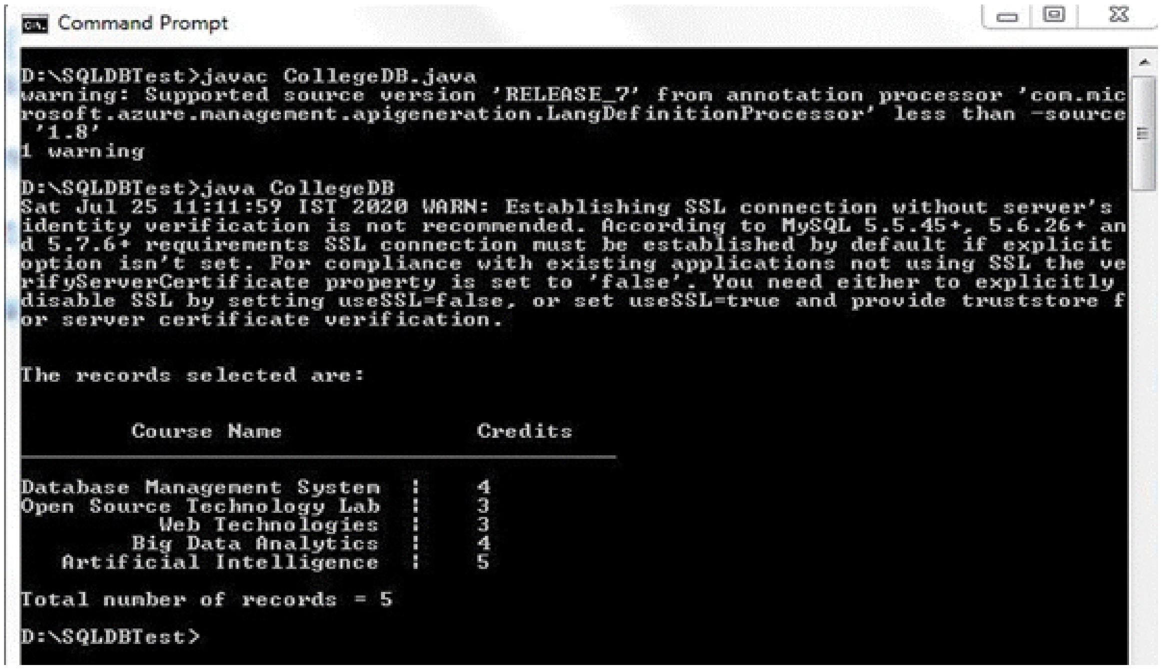
Example: **`con.close();`** as shown in *Figure 4.37*, the output for which is shown in *Figure 4.38*:

Java Code:

```
import java.sql.*;
class CollegeDB{
public static void main(String args[])
{
   try{
            Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection ("jdbc:mysql://
localhost:3306/enggcollegeDB", "root", "root");
// The format is: "jdbc:mysql://hostname:port/databaseName", "username",
"password"
 Statement stmt=con.createStatement()
ResultSet rs = stmt.executeQuery("SELECT Cname, Credits FROM COURSE
WHERE DID =1");
              System.out.println("\n\nThe records selected are:\n");
        int rowCount = 0;
              System.out.println();
              System.out.println("\tCourse Name\t\t Credits");
              System.out.println("__________________________________");
               System.out.println();
                         while(rs.next()) {

                   String Course = rs.getString("Cname");
                   int   Credit   = rs.getInt("Credits");
                   System.out.format("%26s  |%5s",Course,Credit);
                   System.out.println();
                   ++rowCount;
                             }
              System.out.println();
         System.out.println("Total number of records = " + rowCount);
      con.close();
}catch(Exception e){ System.out.println("Driver not found");}
}
}
```

Figure 4.37: *Embedded SQL: Java Code*



```
D:\SQLDBTest>javac CollegeDB.java
warning: Supported source version 'RELEASE_7' from annotation processor 'com.mic
rosoft.azure.management.apigeneration.LangDefinitionProcessor' less than -source
 '1.8'
1 warning

D:\SQLDBTest>java CollegeDB
Sat Jul 25 11:11:59 IST 2020 WARN: Establishing SSL connection without server's
identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ an
d 5.7.6+ requirements SSL connection must be established by default if explicit
option isn't set. For compliance with existing applications not using SSL the ve
rifyServerCertificate property is set to 'false'. You need either to explicitly
disable SSL by setting useSSL=false, or set useSSL=true and provide truststore f
or server certificate verification.

The records selected are:

        Course Name              Credits
-------------------------------------------
Database Management System  :     4
Open Source Technology Lab  :     3
          Web Technologies  :     3
        Big Data Analytics  :     4
   Artificial Intelligence  :     5

Total number of records = 5

D:\SQLDBTest>
```

Figure 4.38: *Embedded SQL*

Conclusion

The Structured Query Language is used to communicate with the relational databases designed by *Donald D. Chamberlin* and *Raymond F. Boyce*.

The SQL commands can be divided into various categories such as DDL, DML, DCL, TCL, and so on. The DDL commands allow you to create the databases, tables, views, and indexes, whereas the DML commands allow you to add, modify, and delete the data from the database.

SQL has the basic data types such as `CHAR`, `VARCHAR`, `INTEGER`, `DATE` etc., to store the values into the database. A special column value NULL is used to represent a missing value or denote unknown values.

SQL offers a rich set of constraints to maintain the data integrity. Entity integrity, referential integrity, domain integrity, check, and unique constraints are examples of some constraints supported by SQL.

To retrieve the data, a `SELECT` statement is used. The SQL query has a `SELECT`, `FROM`, and a `WHERE` clause. The `WHERE` clause is used to express the condition or to filter the rows from relation(s).

SQL supports five aggregate commands `COUNT, SUM, AVERAGE, MAX`, and `MIN`. The Group by clause can be used to group the result with aggregate functions/commands. The Having clause can be used to filter the group-based result. The where and `Having` clause can be used in the same query.

Views are used to provide security by displaying only the selected data. It represents the custom output as if the data were coming from one single table.

SQL provides the string pattern matching capabilities through various string operations.

Trigger is a procedure used to implement the business rule which is beyond the scope of the constraints provided by SQL. It has two types – row-level trigger and statement-level trigger. It is also referred to as event-condition-action. The users must use the triggers carefully to avoid the cascading effect of triggers.

The SQL Joins clause is used to combine the records from two or more tables in a database. They are categorized as Inner Join and Outer Join. Natural join, EQUI join, and Theta join are different types of Inner Joins. Outer Joins can be classified as Left Outer Join, Right Outer Join, and Full Outer Join.

Embedded SQL shows how the SQL statements can be embedded within the higher-level programming languages such as Java.

The table is the basic building block of database design. As we have seen in the previous chapter, the table's structure is of great interest. Even though the Entity Relationship (ER) modeling yields good table structures, it is possible to create poor table structures even in a good database design.

In the next chapter, we will discuss the normalization process which is followed to recognize a poor table structure and to avoid different anomalies.

Questions

1. What are the Advantages of SQL?
2. List the five data types that are used most frequently in the creation of SQL relations.
3. How can the entity integrity and referential integrity constraints be implemented in SQL?
4. List the DDL commands and describe them in short.

5. What is the view? How is it created and stored? What are the benefits and limitations of view?
6. Explain the following with proper examples:

   a) Create statement

   b) Insert statement

   c) Delete statement

   d) Drop statement

   e) Alter statement

   f) Data control language (DCL)

   g) Data manipulation language (DML)

   h) Transaction control language (TCL)

   i) Data definition language (DDL)
7. What is the difference between the DELETE and DROP commands?
8. List and explain the various features provided by SQL.
9. What is NULL in SQL? How are the null values handled in SQL?
10. What are the different aggregate functions in SQL? Explain with the help of examples.
11. List the aggregate functions and justify the need for any two aggregate functions.
12. Explain Triggers with suitable examples.
13. What are SQL indexes? Explain the types of indexes with the help of examples.
14. Define and explain Nested Queries.
15. Write a short note on Cursors and its types.
16. Explain Joins and the types of Joins with suitable examples.
17. Explain programming with JDBC.
18. What is the need of embedded SQL? Describe the scenario in which you would choose to use the embedded SQL over SQL.
19. Consider the following relations scheme:

    Employee (<u>eid</u>, ename, age, salary)

    Works (<u>eid</u>, <u>did</u>, pct-time)

Department (did, budget, managerid)

(Employees can work in more than one department and pct-time gives % of time in that dept.)

Now, write the following queries in SQL:

a) Print name and salary of employees whose salary exceeds the budget of all departments where he/she works.

b) Find the names of managers who manage most number of departments.

c) Find the top 10 earning employees.

d) Delete all employees who do not put in 25% work in at least one department in which they work.

20. Consider the following Company schema:

Employee (employee-name, street, city)

Works (employee-name, company-name, salary)

Company (company-name, city)

Manages (employee-name, manager-name)

Now, write the following queries:

a) Manager earns the maximum salary than any employee.

b) A manager lives in the same city in which the company is located.

21. Consider the Company schema, given in Question 20, and write the following queries:

a) Find the names of all employees who work for Vaidehi Enterprises.

b) List the employees whose salary is greater than 50000 and lives in Navi Mumbai.

c) Find all employees in the database who do not work for Vaidehi Enterprises and lives in Pune.

d) Find the average salary of all employees working for Vaidehi Enterprises and having more salary than that of every employee of Lux Corporation.

e) Find the company that has the maximum employees.

22. Write the SQL code that will create the following STUDENT relation:

STUDENT (Rollno, Fname, Sname, Emailid, ContactNo)

a) No duplicate values for Rollno; is used to uniquely identify the student from the student set.

b) The student may have more than one contact number, but a single email id assigned by college.

c) Now, consider the students' addresses having the values of Street, City, and PIN that also need to be stored in the database. Change the schema to accommodate the student's address.

d) The student's age needs to be computed without storing it directly; modify the student schema.

e) Modify the schema if each student enrolls for a course which has a unique courseid and registers him/herself in a single department.

23. Write appropriate DDL statements to create a LIBRARY database with Book, Author, Publisher, and Student who borrows a book from the Library. Show the primary key, foreign key, and other unique constraints which may be enforced during the creation of various relations.

24. With reference to *Figure 4.39*, list the relations and give equivalent SQL definitions for the same:

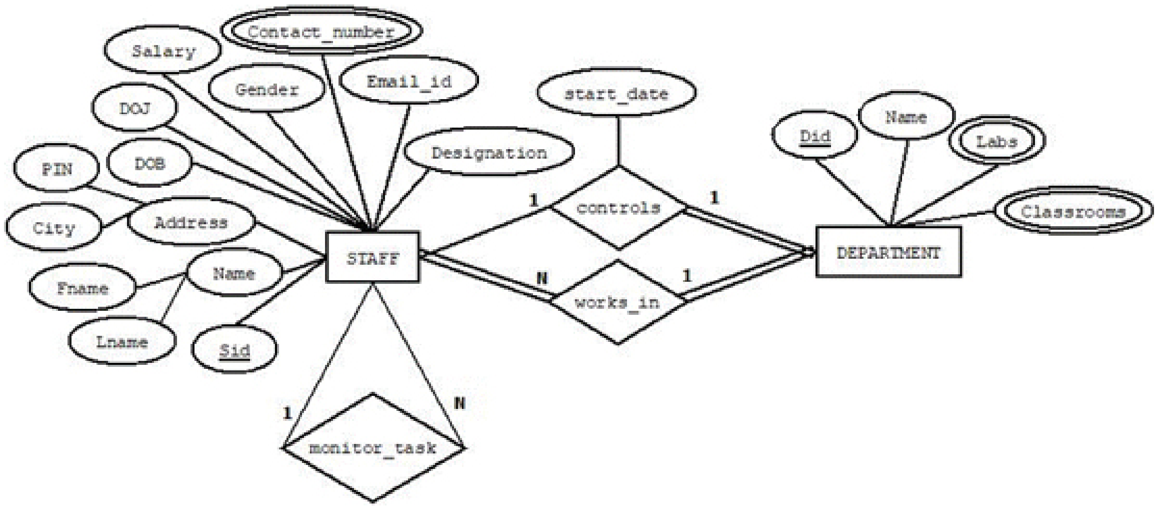


Figure 4.39: *STAFF-DEPARTMENT Relationship*

25. Consider the following schema for the Engineering College Library:

STUDENT (<u>Rollno</u>, Name, Branch)

BOOK (<u>ISBN</u>, Title, Author, Publisher)

ISSUE (<u>Rollno</u>, <u>ISBN</u>, IssueDate)

Write the SQL queries for the following:

a) List the Rollno and Names of all the IT students.

b) Find the names of the students who have issued the book published by 'BPB Publications'.

c) List the Title and Authors of all the books published by 'BPB Publications'.

d) List the Titles and numbers of books of 'BPB Publications' issued before December 2020.

26. What is a Trigger? List the advantages and disadvantages of Triggers.
27. With specific use cases, explain the concept of Triggers?
28. What are the differences between Trigger and Stored Procedure?

CHAPTER 5
Relational Database Design

Introduction

SQL was co-developed at IBM by *Raymond F. Boyce* alongside *Donald D Chamberlin* in the early 1970s and was called *SEQUEL*; it was based on their original language called SQUARE. SEQUEL was designed to manipulate and retrieve the data in relational databases, and *Boyce* published SEQUEL, a *Structured English Query Language*, which detailed their refinements to SQUARE and introduced us to the data retrieval aspects of SEQUEL.

It was one of the first languages to use Edgar F Codd's relational model. SEQUEL was renamed as SQL by dropping the vowels; SQL has become the most used relational database language.

Normalization is a very important step in database design. After normalization, the database systems would be accurate, and you can have quick access to your data without compromising the integrity of data storage. The goal of the relational database design is to generate a set of database schemas that store the information without unnecessary redundancy.

The Boyce-Codd normal form (BCNF) was developed by Boyce and Edgar F Codd. It is a type of normal form that is used in database normalization.

Boyce-Codd accomplishes the goal of Normalization and allows the users to retrieve the information, based on the functional dependencies. The BCNF removes redundancy and it is stronger version of the third normal form.

Structure

In this chapter, we will cover the following topics:

- Pitfalls in the relational database design
- Normalization process and its importance in the database design process
- Various types of normal forms such as 1NF, 2NF, 3NF, BCNF, and 4NF
- Need to apply denormalization in some cases for generating information efficiently

Objectives

After studying this chapter, you will be able to understand functional dependency and functional decomposition, apply a normalization process which reduces the database anomalies from the relational table, organize the data into logical groups, and minimize the duplicated data stored in a database. You will also be able to recognize and identify the use of normalization and functional dependency and perform normalization based on the functional dependency.

Introduction

In Chapter 2, *Entity Relationship (ER) Modeling*, we learned how to develop a conceptual design for the database. Then, in Chapter 3, *Relational Model & Relational Algebra*, we learned how to convert the ER models into relational models. It is possible to create poor relational structures even from a good database design. Redundant data and database anomaly problems may occur due to bad database design. It is difficult to work with such databases. To avoid such situations, one must apply the normalization process. It is the process of efficiently organizing the data in a database, ensuring that the data is logically stored, and that accessing it is quick and efficient.

Normalization works through a series of stages called normal forms. The first three stages are described as first normal form (1NF), second normal form (2NF), and third normal form (3NF). From a structural point of view, 2NF is better than 1NF, and 3NF is better than 2NF. For most purposes in the business database design, 3NF is as high as you need to go in the normalization process. However, you will discover that properly designed 3NF structures also meet the requirements of the fourth normal form (4NF).

Quality data is the requirement of today as it plays a key role in strategic decision-making for the industry, government, and academia. Traditionally, the data resources have been managed by a file processing system, but it has its own limitations as we discussed in Chapter 1, *Database System Concepts and Architecture*. During information processing within an organization, the data resources are typically analyzed in the form of a data model such as database management systems (DBMS). The analyzing and designing of the database should obey certain rules of good behavior, otherwise it results in undesirable properties called data anomalies. These anomalies often lead to repetition of information, inability to represent certain information, and loss of information, and in such cases, normalization is the only viable solution.

The normalization theory of relational databases dates back to *E.F. Codd's* first seminal papers about the relational data model (*Codd*, 1970). Normalization is the process of grouping the data into well-defined structures. Normalization is, in the relational database design, the process of organizing the data to minimize redundancy. It usually involves dividing a database into two or more tables and defining the relationships between the tables. The objective is to isolate the data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.

Pitfalls in relational-database design

Any relational database design allows us to store the information, as well as retrieve the information easily and quickly.

However, we can have good and bad designs. A bad design may have several properties, including the following:

- Repetition of information
- Inability to represent certain information
- Loss of information
- Inability to extract the information on time

The preceding properties are a few common design pitfalls that can harm a database system. So, now let’s see what the various database design practices are, which may lead to a bad design.

Poor design/planning

Poor design may lead to problems related to data redundancy and data integrity. It is impossible to predict everything in advance that the proposed design will fulfill, but it is better to mitigate against the potential problems by careful planning.

Ignoring normalization

In the process of normalization, it breaks down the tables until each table represents and describes one and only one Entity. Normalization basically identifies the anomalies in a database design. If the designer ignores the normalization step, it will later cause problems when the data is queried, inserted, or updated.

So, normalizing your database design is essential for good performance and ease of development. In general, all database designs need to be normalized at least up to the 3rd Normal Form.

Poor naming standards

Choosing the names for tables and columns in a database design is often neglected but it can have a considerable impact on its usability and for future reference.

Always choose meaningful names for both the table and the column names and also ensure that consistency is maintained throughout the system.

Lack of documentation

Documentation is essential to quality and process control. It encourages knowledge sharing, which empowers the team members to understand how the processes work.

Assume, in the organization, there is an employee who knows everything about the project but in case of his absence, there’s no one to tackle the problems. The organizations using documentation shares knowledge, and benefits by reducing re-working from scratch that wastes your precious time.

Not considering SQL facilities

Designers often choose not to consider the SQL facilities, such as stored procedures and integrity checks, during the design stages of a project as they are not necessary at the design stage; however, it can help avoid problems at a later stage and protect data integrity.

Lack of testing

Failure to test a database design with a sample of real data can cause serious problems in a database system. Generally, the relational database design is started from an abstract level, using the modeling tools to reach at a design. The drawback to this process is that many a times it will not relate with the actual data.

Design guidelines for relational schema

Before discussing the normalization process in detail, let's discuss the four informal guidelines that may be used as measures to determine the quality of the relation schema design, as follows:

Guideline 1: Semantics of relation attributes should be easy

In a relation, each tuple should explain one entity or one relationship instance. It is not a good practice to combine the attributes from multiple entity types and relationship types into a single relation. Only the foreign keys should be used in a relation when referring to other relations.

Design a schema in such a way that all the relations can be explained easily. The semantics of the attributes should be easy to understand the meaning.

Example:

Staff_Dept (Sid, Sname, Phone, Gender, Did, Dname, Head_id, Start_date)

The preceding schema is an example of bad design as the attributes of two different Entities are mixed in one relation. The correct way to represent the same information is as follows:

Staff (Sid, Sname, Phone, Gender, Did)
Dept (Did, Dname, Head_id, Start_date)

Guideline 2: Redundant information in tuples and update anomalies

Design a schema that does care about the insertion, deletion, and update anomalies. If there are any anomalies present, then the applications should be made to take them into account.

Example:

Staff_Dept (Sid, Sname, Phone, Gender, Did, Dname, Head_id, Start_date)

The preceding schema is an example of a bad design as it introduces insert, delete, and update anomalies. Let us understand it with the help of the following examples:

Insert anomaly:

Consider the preceding schema; here, one cannot insert the Staff details unless any department is assigned to him/her. Similarly, the Department details cannot be inserted unless any Staff is assigned to it.

Delete anomaly:

Consider the scenario where a department is deleted; here, all the staff member's details will be deleted who are working in that department; as a result, many staff details would be lost.

Update anomaly:

Consider the situation where we want to update the department name from "*Comp*" to "*Computer*" where 80 Staff is currently working. As a result, it may cause the updates to be made for all 80 staff members working in Department "*Comp*".

Guideline 3: Null values in tuples

Avoid having such attributes in a base relation whose values may frequently be NULL.

There are many reasons for having null values in the relation such as attribute not applicable, value unknown, or value known but unavailable. Placing unnecessary Null values in a relation may waste the storage space, and also there would be problems with understanding the meaning of the attributes.

For example, if only 10 percent of the staff have won the award, there is little justification for including an attribute Award in the Staff relation; rather, a relation `Staff_Award(Sid, Award_Title)` can be created to include the tuples for only the staff who won the award.

Guideline 4: Spurious tuples

The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations. You can achieve this by applying the join operation on the primary key and foreign key pair attributes.

Functional dependencies

Functional Dependencies (**FD**) basically helps maintain the quality of the database design. It is denoted by an arrow "→".

It is a relationship that exists between two attributes (generally, primary key and non-key attribute within a table). The following is an example:

The functional dependency of X on Y is represented by $X \rightarrow Y$.

Here, X is known as a determinant, and Y is known as a dependent.

Armstrong's Axioms property of functional dependency

Armstrong's Axioms property was developed by *William Armstrong* in 1974. Armstrong's Axioms are used to conclude the functional dependencies on a relational database. The inference rule is a type of assertion. It can apply to a set of FDs to derive another FD. Using the inference rule, we can derive additional functional dependencies from the initial set.

The functional dependency has six types of inference rules, which are as follows:

(1) **Reflexive rule**

If A ⊇ B, then A → B { if B is a subset of A, then A determines B}

(2) **Augmentation rule**

If { A→B} , then AC →BC and that is why the augmentation is also called a partial dependency.

(3) **Transitive rule**

In the transitive rule, if A determines B and B determines C, then A must also determine C, as shown as follows:

{ A → B, B → C} , then { A → C}

(4) **Union rule**

The union rule says, if A determines B and A determines C, then A must also determine B and C, as shown as follows:

{ A → B, A → C} , then { A → BC}

(5) **Decomposition rule**

It is the reverse of the union rule. This rule says, if A determines B and C, then A determines B and A determines C separately, as shown as follows:

{ A → BC} , then { A → B, A → C}

(6) **Pseudo transitive rule**

In the Pseudo transitive rule, if A determines B and BC determines D, then AC determines D, as shown as follows:

{ A → B, BC → D} , then { AC → D}

Types of functional dependency

The following are the different types of functional dependencies:

- **Multivalued functional dependency**

  It occurs when more than one independent attribute exists with multiple values in the same table. This can be represented as follows:

  A -> B

  A -> C

  A -> D

Here A, B, C, and D are the attributes of the same table, where A is the primary key, and B, C, and D are the non-key attributes. Here B, C, and D are functionally dependent on A, and not dependent on each other. In this example, these three columns (B, C, D) are said to be multivalue dependent on column A.

- **Trivial functional dependency**

  The dependency of an attribute on a set of attributes is known as a trivial functional dependency if the set of attributes includes that attribute, as shown as follows:

  A->B is a trivial functional dependency if B is a subset of A.

- **Non-trivial functional dependency**

  The functional dependency which is also known as a non-trivial dependency, when A->B holds true, where B is not a subset of A. In a relationship, if attribute B is not a subset of attribute A, then it is considered a non-trivial dependency.

Need for Normalization

Normalization helps simplify the database design to achieve the optimal structure consisting of the atomic values. Through this process, we can reduce the redundancy and improve data integrity.

Assume the following Author's database which holds the Author's Name, Book's Id, Title of Book , DateOfPublish, and Type of Book, as shown in *Table 5.1*:

Book_Id	Author_name	Book_Title	Publish_date	Book_Type
1	Humairah	Education and Career	20-Sep-1986	Education
2	Bushra	Art of Living	12-Feb-1981	self-development
3	Bushra	Secret of Success	03-Mar-1970	self-development
4	Sneha	Healthy Diet	13-10-1983	Health sector
5	Bushra	Mantra for Leadership	13-Sep-1984	self-development

Table 5.1: *Author's database*

The *Table 5.1* shows the denormalized database, may be useful but can have some shortcomings when inserting, updating, or deleting data. It can result in the following three anomalies:

Update anomaly

If we need to update an author's name, we’ll need to update multiple rows (in this case, at 3 places). It is the same with the book type (in this case, also at 3 places). This could result in errors. If we update some rows, but not others, we’ll end up with inaccurate data. This is known as an update anomaly.

Assume if we need to update the author's name from Bushra to “*Bush*”; we will need to update at three different places. So, if the user made the changes in the 2nd and 3rd row and not in the 5th row, there will be a mismatch and will be incorrect data in the database.

Insertion anomaly

There is a possibility that some author’s book has not yet been published. In such a situation, we can add the author's detail, but a few fields need to be set as NULL. This is known as an insertion anomaly.

When the Book has been released, we might add a new record with complete information, but in this case, we will end up with two rows for that Author (one with fewer details and another with all the details), and it could create confusion.

Deletion anomaly

If we need to delete a piece of information regarding a particular book, we can’t do that without deleting the author. If an author has written only one book, and if we delete that book, we’ll end up deleting the author’s information from the database. We’ll no longer have any record of that author in the database. This is known as a deletion anomaly.

Levels of Normalization

Normalization has various levels, where each level depends on the previous level. The most basic level of normalization is the 1NF, followed by the 2NF, and so on.

The majority of the transactional databases can be found in the 3NF.

For any database to satisfy a given level, it must satisfy the rules of all the lower levels as well as the rules for the given level. For example, for any database that needs to be in 3NF, it must be in 1NF, 2NF, and also 3NF.

Unnormalized Form (UNF)

A database is in UNF if it has not been normalized at all.

If a table has non-atomic values, it is said to be UNF. The non-atomic values can be further decomposed and simplified.

First Normal Form (1 NF)

This form disallows composite attributes, multivalued attributes, and nested relations, that is, the attributes whose values for an individual tuple are non-atomic.

Second Normal Form (2 NF)

A relation schema R is in 2NF if the relation is in 1NF and if every non-prime attribute A in R is fully functionally dependent on the primary key.

Third Normal Form (3 NF)

A relation schema R is in 3NF if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key.

Boyce-Codd Normal Form (BCNF)

A relation schema R is in Boyce-Codd Normal Form (BCNF) if every determinant is a candidate key.

OR

A relationship is said to be in BCNF if it is already in 3NF and the left-hand side of every dependency is a candidate key.

Fourth normal form (4 NF)

A relation is said to be in 4NF if each table contains no more than one multivalued dependency per key attribute.

Fifth normal form (5 NF)

A relation is in 5NF if it is in 4NF and does not contain any join dependency and the joining should be lossless.

Normalization process

Let us consider that the faculty information is maintained department-wise in an engineering college, as shown in *Table 5.2*. From the table, it becomes apparent that the faculty may work in more than one department. The qualification decides the faculty type which, in turn, will play a role in deciding the payment per session. The qualification and faculty type also decide the weekly load. If we assume the manual maintenance of the records, then it may result in data redundancy as the same faculty members are working in more than one department, i.e., the faculty record appears at multiple instances. The manual maintenance of the records may also result in data inconsistency if the record of the faculty is not updated for all the occurrences of the record. For example, if *Mr. Mustafa Ahmed* completes his Ph.D., then it must be updated in both the departments in which he is working, otherwise it may result in inconsistency. Similarly, the other anomalies can be identified and highlighted. This means that the structure in *Table 5.1* is not suitable to maintain the data. It must follow the guidelines of *E.F. Codd*. Based on these guidelines, we have the following observations:

- The DeptID seems to be the best attribute to serve as the primary key or at least a part of the primary key but some records have DeptID as null.

- The table entries invite data inconsistencies. For example, the FacultyType value 'Part Time' might be entered as 'PT' in some cases and the qualification as 'Bachelor of Engineering' instead of 'BE'.
- The table displays data redundancies. Those data redundancies yield the following anomalies:
    - If the department and faculty are not separate relations/tables, then during the insertion of new faculty, it becomes mandatory to assign the faculty to a particular department, or a new department must be created to complete the faculty data entry.
    - If both the faculties, *Mr. Raj Vartak* and *Ms. Tanuja*, leave the department, and if the faculty data is deleted, the department information will also be deleted. To prevent the loss of department information, a fictitious faculty must be created just to save the department information.
    - Even if a new faculty is assigned to the department, then unnecessarily the associated attribute values need to be entered and can be decided based on rules. For example, if the new staff joins the civil department with a Ph.D. qualification, then he should not have more than 8 hours of load.
    - Suppose that, if the '*Part Time*' Faculty type is given a new name such as '*Adhoc',* then it has to be changed at every place where it occurs, otherwise it will result in an inconsistent state.

Refer to *Table 5.2* that displays department-wise faculty information:

Dept ID	DeptName	Faculty ID	FacultyName	Qualification	FacultyType	Payment	Weekly Load
101	Computer Engineering	F01	Dr. Krishna Prakash	PhD	Adjunct professor	2000	6
	Computer Engineering	F02	Mr. Mustafa Ahmed	ME	Part-Time	1000	12
	Computer Engineering	F03	Dr. Rupa Chabra	PhD	Adjunct professors	2000	8
102	Information Technology	F01	Dr. Krishna Prakash	PhD	Adjunct professors	2000	6
	Information Technology	F02	Mr. Mustafa Ahmed	ME	Part-Time	1000	12
	Information Technology	F10	Mr. Yogesh Patil	ME	Part-Time	1000	16

Dept ID	DeptName	Faculty ID	FacultyName	Qualification	FacultyType	Payment	Weekly Load
103	Electronics Engineering	F25	Dr. Neeraj	PhD	Adjunct professors	2000	8
	Electronics Engineering	F26	Mr. G Pathak	ME	Part-Time	1000	12
104	Civil Engineering	F13	Mr. Raj Vartak	ME	Part-Time	1000	12
	Civil Engineering	F17	Ms. Tanuja	BE	Hourly Basis	500	12
105	Electronics and Communication Engineering	F14	Dr. Raj Goswami	PhD	Adjunct professors	2000	4
	Electronics and Communication Engineering	F25	Dr. Neeraj	PhD	Adjunct professors	2000	6
	Electronics and Communication Engineering	F27	Ms. Neena	BE	Hourly Basis	500	12

Table 5.2: *The department-wise faculty information*

Now that we have found the drawbacks of *Table 5.2* through our observation, let us try to overcome them through the process of normalization.

Conversion to 1NF

1NF states that there should be no repeating groups and the primary keys should be identified. Let us understand the repeating groups. In *Table 5.2*, four entries are corresponding to the computer engineering department. Similarly, the Information Technology department has three record entries. If a new faculty joins the Computer Engineering Department, then the number of entries in this group will be five, i.e., in repeating groups, the addition of a new record leads to an increase in the number of entries by one. This is a good example of data redundancy, and to remove such data redundancy, repeating groups must be eliminated.

Elimination of repeating groups is the first step toward the three-step process to achieve the first normal form. The next step is to identify the primary key and the final step is to identify all the dependencies. These dependencies help us identify as to which normal form is the table in.

1. **Repeating groups elimination**

   For this, each record should define a single entity. All the cells should contain a single value. The cells that have null values must be replaced by appropriate data values. After making such changes, *Table 5.3* is obtained, which is in the first normal form:

Dept ID	DeptName	Faculty ID	FacultyName	Qualification	FacultyType	Payment	Weekly Load
101	Computer Engineering	F01	Dr. Krishna Prakash	PhD	Adjunct professor	2000	6
101	Computer Engineering	F02	Mr. Mustafa Ahmed	ME	Part-Time	1000	12
101	Computer Engineering	F03	Dr. Rupa Chabra	PhD	Adjunct professors	2000	8
102	Information Technology	F01	Dr. Krishna Prakash	PhD	Adjunct professors	2000	6
102	Information Technology	F02	Mr. Mustafa Ahmed	ME	Part-Time	1000	12
102	Information Technology	F10	Mr. Yogesh Patil	ME	Part-Time	1000	16
103	Electronics Engineering	F25	Dr. Neeraj	PhD	Adjunct professors	2000	8
103	Electronics Engineering	F26	Mr. G Pathak	ME	Part-Time	1000	12
104	Civil Engineering	F13	Mr. Raj Vartak	ME	Part-Time	1000	12
104	Civil Engineering	F17	Ms. Tanuja	BE	Hourly Basis	500	12
105	Electronics and Communication Engineering	F14	Dr. Raj Goswami	PhD	Adjunct professors	2000	4

DeptID	DeptName	FacultyID	FacultyName	Qualification	FacultyType	Payment	WeeklyLoad
105	Electronics and Communication Engineering	F25	Dr. Neeraj	PhD	Adjunct professors	2000	6
105	Electronics and Communication Engineering	F27	Ms. Neena	BE	Hourly Basis	500	12

Table 5.3: *Table in 1NF*

2. **Primary key identification**

   Once the repeating groups have been eliminated, the next step is to identify the attribute that can be used to determine each row uniquely. As obvious, only DeptID cannot serve as the primary key because it will result in multiple rows corresponding to the respective departments. For example, DeptID=104 will correspond to both the faculties viz. *Mr. Raj Vartak* and *Ms Tanuja*. In other words, DeptID alone does not identify each row uniquely. Nor is any other attribute of *Table 5.3* capable enough to identify each row uniquely.

   Hence, a combination of more than one attribute can serve as the primary key. In *Table 5.3*, DeptID, along with the attribute FacultyID, uniquely identifies each row. For example, DeptID=104 and FacultyID=F13 gives a record of *Mr. Raj Vartak* distinctively along with his qualification and other details. Hence, a combination of DeptID and FacultyID is identified as the primary key for *Table 5.3*.

3. **Dependencies identification**

   Now that the primary key has been identified, the next step is to identify the dependencies. Since (DeptID, FacultyID) is the primary key, it means that these two attributes determine the values of the other attributes in the table, i.e., DeptName, FacultyName, Qualification, FacultyType, Payment, and WeeklyLoad. This can be represented as follows:

   DeptID, FacultyID -> DeptName, FacultyName, Qualification, FacultyType, Payment, WeeklyLoad

   This is also referred to as functional dependency. It means that DeptName, FacultyName, Qualification, FacultyType, Payment, and WeeklyLoad are functionally dependent on (DeptID, FacultyID).

Let's take another example for functional dependency. DeptID corresponds to a particular department. For example, DeptID=101 is for the computer engineering department, whereas DeptID=102 is for information technology and so on. In other words, DeptID functionally determines DeptName. This can also be represented as follows:

DeptID -> DeptName

It denotes that DeptName is fully functionally dependent on DeptID, where each value of DeptID determines one and only one value of DeptName. Here, DeptID is the '*determinant'* attribute and DeptName is the 'dependent' attribute.

Note that in step 2, we identified a composite key (DeptID, FacultyID) to serve as the primary key. Therefore, a primary key should be able to determine each row uniquely, i.e., (DeptID, FacultyID) -> DeptName. But as already discussed earlier, DeptName can be determined by DeptID alone, which is a subset of the composite keys. Therefore, we conclude that DeptName is not fully functionally dependent on the composite key (DeptID, FacultyID). This is better referred to as partial dependency, because the DeptName attribute is partially dependent on the Composite key (DeptID, FacultyID) as DeptId (which is a subset of the composite key) uniquely identifies DeptName.

Similarly, FacultyName, Qualification, FacultyType, and Payment can be determined using FacultyID alone, as shown as follows:

FacultyID -> FacultyName, Qualification, FacultyType, Payment

This is another example of partial dependency. Here, (DeptID, FacultyID) is the primary key but a subset of the composite key FacultyID uniquely determines FacultyName, Qualification, FacultyType, and Payment attributes.

Other functional dependencies that can be easily identified are that FacultyID determines the Qualification of a faculty which, in turn, determines the FacultyType. This can be represented as follows:

FacultyID -> Qualification

Qualification -> FacultyType

Hence,

FacultyID -> FacultyType

Here, FacultyID determines the value of FacultyType via Qualification. This is a classic example of transitive dependency. Please note that in Qualification -> FacultyType, both the determinant and dependent are non-prime attributes. Therefore, whenever there is functional dependency among the non-prime attributes, it's an indication that transitive dependency exists in the relation.

Also, FacultyType determines Payment, as shown as follows:

FacultyID -> Qualification

Qualification -> FacultyType

FacultyType -> Payment

Hence,

FacultyID -> Payment

FacultyID determines the value of Payment via Qualification and FacultyType. This is another example of transitive dependency. Here too, both the determinant and dependent are non-prime attributes in the functional dependencies; Qualification -> FacultyType and FacultyType -> Payment. Hence, another transitive dependency exists in the relation, as shown in *Figure 5.1*:

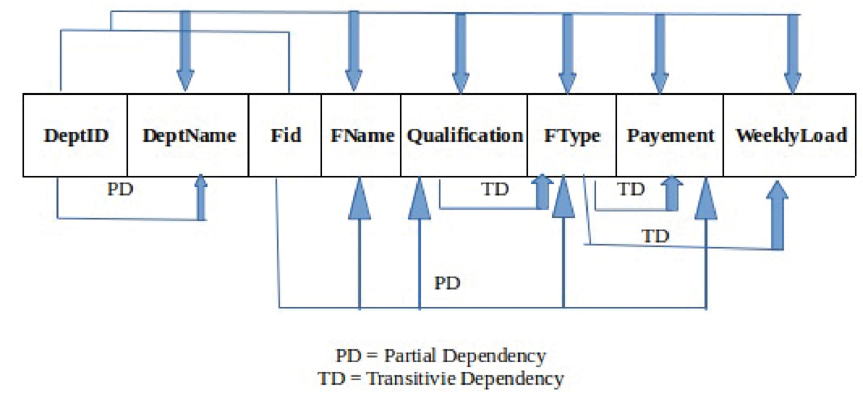


Figure 5.1: *Dependency diagram for Table 5.3*

All the dependencies of a relation, as discussed earlier, is easily illustrated with the help of a dependency diagram, as shown in *Figure 5.1*.

1NF (DeptID, FacultyID, DeptName, FacultyName, Qualification, FacultyType, Payment, WeeklyLoad).

Partial dependencies:

DeptID -> DeptName

FacultyID -> FacultyName, Qualification, FacultyType, Payment

Transitive dependencies:

Qualification -> FacultyType

FacultyType -> Payment

To remove partial dependencies and transitive dependencies, we must move to the higher normal forms.

Conversion to 2NF

A table is in 2NF when it is in 1NF and has no partial dependency. A need to convert a table from 1NF to 2NF exists only when the primary key is a composite key resulting in partial dependencies. Therefore, to convert a table to 2NF, the partial dependencies should be eliminated by creating new tables and reassigning the corresponding dependent attributes.

The following are the steps:

Step 1: Elimination of partial dependencies by creating new tables

We already identified two partial dependencies, as depicted in *Figure 5.1*. Since the determinant in those two partial dependencies is a subset of the composite key, create a new table with the determinant as the primary key. Therefore, two new tables are created, one with the primary key as DeptID, which is named as Department, and another one with the primary key as FacultyID, which is named as Faculty. Both these attributes act as the foreign keys in the original table which is named Workload. Please note that the original table is also retained where the composite key (DeptID, FacultyID) is the primary key.

Step 2: Reassignment of dependent attributes

Once the primary key for the new tables has been identified, the next step is to determine the corresponding dependent attributes. These attributes can be easily identified from the partial dependencies depicted in *Figure 5.1*. For example, DeptName is the dependent attribute for the Department table having the primary key as DeptID. Note that the DeptName attribute is removed from the original table Workload. Similarly, FacultyName, Qualification, FacultyType, and Payment are the dependent attributes for the Faculty table having the primary key as FacultyID. These dependent attributes are also removed from the original table Workload. Please note that only the dependent attributes are removed from the original table, whereas the determinant attributes are retained in the original table. Thus, a total of three tables, viz. Department, Faculty, and Workload, are there in 2NF which are described by the relational schema, depicted as follows:

Department (DeptID, DeptName)

Faculty (FacultyID, FacultyName, Qualification, FacultyType, Payment)

Workload (DeptID, FacultyID, WeeklyLoad)

Consequently, the three tables in 2NF with the corresponding entries are shown in *Table 5.4*, *Table 5.5*, and *Table 5.6*:

DeptID	DeptName
101	Computer Engineering
102	Information Technology
103	Electronics Engineering
104	Civil Engineering
105	Electronics and Communication Engineering

Table 5.4: *Department*

FacultyID	FacultyName	Qualificati on	FacultyType	Payme nt
F01	Dr. Krishna Prakash	PhD	Adjunct professor	2000
F02	Mr. Mustafa Ahmed	ME	Part-Time	1000
F03	Dr. Rupa Chabra	PhD	Adjunct professors	2000
F10	Mr. Yogesh Patil	ME	Part-Time	1000
F25	Dr. Neeraj	PhD	Adjunct professors	2000
F26	Mr. G Pathak	ME	Part-Time	1000
F13	Mr. Raj Vartak	ME	Part-Time	1000
F17	Ms. Tanuja	BE	Hourly Basis	500
F14	Dr. Raj Goswami	PhD	Adjunct professors	2000
F27	Ms. Neena	BE	Hourly Basis	500

Table 5.5: *Faculty*

DeptID	FacultyID	WeeklyLoad
101	F01	6
101	F02	12
101	F03	8
102	F01	6
102	F02	12
102	F10	16
103	F25	8
103	F26	12
104	F13	12
104	F17	12
105	F14	4
105	F25	6
105	F27	12

Table 5.6: *Workload*

At this stage, the partial dependencies have been removed. Let us revisit the anomalies discussed at the start of this chapter, as follows:

a) Now, if a new faculty joins the college, then an entry can be made in the faculty table without assigning him/her any particular department and weekly load. Thus, the insertion anomaly is removed.

b) If both the faculties, *Mr. Raj Vartak* and *Ms. Tanuja*, leave the department and if the faculty data is deleted, still the Civil Engineering department information is very much intact in the Department table. Thus, the deletion anomaly is removed.

c) Suppose that, if the '*Part Time*' Faculty type is given a new name, such as '*Adhoc*', then it must be changed at every place wherever it occurs, otherwise it will result in an inconsistent state. This updation anomaly is yet to be removed. The reason is, the relational schema still contains the transitive dependency.

Now, to remove the transitive dependencies, we move to the third normal form.

Conversion to 3NF

A table is in 3NF when it is in 2NF and has no transitive dependency. A need to convert a table from 2NF to 3NF exists only when the non-prime attributes determine the other non-prime attributes. In such cases, the primary key relies on the non-prime attributes to functionally determine the other non-prime attributes. Similarly to the 2NF, new tables are created in 3NF but to remove the transitive dependencies.

The following are the steps:

Step 1: Elimination of transitive dependencies by creating new tables

We already identified two transitive dependencies in the previous step. For each transitive dependency, a new table is to be created with the determinant in the transitive dependency as the primary key for the respective table. For example, for the transitive dependency Qualification -> FacultyType, a new table Designation is created with the Qualification as Primary Key. Similarly, another table with the name Charges is created having the primary key as FacultyType for the transitive dependency FacultyType->Payment. Please note that the determinant Qualification is retained in the original table, Faculty, in this case, to serve as a foreign key. Also note that even though FacultyType is a determinant for the transitive dependency FacultyType->Payment, it is not retained in the original table Faculty because FacultyType is a dependent attribute in the previous transitive dependency Qualification -> FacultyType and the dependent attributes are removed from the original table during the conversion to 3NF.

Step 2: Reassignment of dependent attributes

Similar to the steps followed during the 2NF conversion, once the new tables have been created and the primary keys have been identified, the next step is to determine the dependent attributes from the transitive dependencies. For example, in the transitive dependency, Qualification -> FacultyType, FacultyType is the dependent attribute for table Designation.

This attribute is, therefore, removed from the original table Faculty. Similarly, in transitive dependency, FacultyType->Payment, Payment is the dependent attribute for the table Charges and hence, removed from table Faculty. Thus, a total of five tables, viz. Department, Faculty, Designation, Charges, and Workload, are there in 3NF which are described by relational schema, as depicted as follows:

Department (DeptID, DeptName)

Faculty (FacultyID, FacultyName, Qualification)

Designation(Qualification, FacultyType)

Charges(FacultyType, Payment)

Workload (DeptID, FacultyID, WeeklyLoad)

Consequently, the five tables in 3NF with the corresponding entries are shown in *Table 5.6*, *Table 5.7*, *Table 5.8*, *Table 5.9*, *Table 5.10*, and *Table 5.11*:

DeptID	DeptName
101	Computer Engineering
102	Information Technology
103	Electronics Engineering
104	Civil Engineering
105	Electronics and Communication Engineering

Table 5.7: *Department*

FacultyID	FacultyName	Qualification
F01	Dr. Krishna Prakash	PhD
F02	Mr Mustafa Ahmed	ME
F03	Dr. Rupa Chabra	PhD
F10	Mr. Yogesh Patil	ME
F25	Dr. Neeraj	PhD
F26	Mr. G Pathak	ME
F13	Mr. Raj Vartak	ME
F17	Ms. Tanuja	BE
F14	Dr. Raj Goswami	PhD

FacultyID	FacultyName	Qualification
F27	Ms. Neena	BE

Table 5.8: *Faculty*

Qualification	FacultyType
PhD	Adjunct professor
ME	Part-Time
BE	Hourly Basis

Table 5.9: *Designation*

FacultyType	Payment
Adjunct professor	2000
Part-Time	1000
Hourly Basis	500

Table 5.10: *Charges*

DeptID	FacultyID	WeeklyLoad
101	F01	6
101	F02	12
101	F03	8
102	F01	6
102	F02	12
102	F10	16
103	F25	8
103	F26	12
104	F13	12
104	F17	12
105	F14	4
105	F25	6

DeptID	FacultyID	WeeklyLoad
105	F27	12

Table 5.11: *Workload*

Now that the table has been converted to 3NF, let us revisit the updation anomaly which was pending after 2NF.

Suppose that, if the '*Part Time*' Faculty type is given a new name, such as '*Adhoc*', then it must be changed only at one place, that is, in the Charges table where FacultyType is serving as a primary key. Note that FacultyType is serving as a foreign key in the table Designation. Hence, due care should be taken to maintain the referential integrity.

Thus, all the anomalies listed at the start of the chapter have been removed by converting the table finally to 3NF.

Boyce Codd Normal Form (BCNF)

Boyce Codd Normal Form (BCNF) is the normal form with respect to the functional dependencies. In fact, all normal forms are with functional dependencies. For example, the fifth normal, which we will discuss in a short time, is based on the concept of join dependencies. It is an advanced (3.5) version of 3NF and stricter than 3NF. The benefits of BCNF are well understood at an intuitive level. BCNF and its predecessor, i.e., the 3NF were introduced by Codd. The main reason to introduce the BCNF is to eliminate the "*anomalous*" update behavior of the relations. BCNF accomplishes this by "*placing independent relationships into independent relations*." For BCNF, the table should be in 3NF, and for every FD, LHS is a super key. Let us define the BCNF formally, as follows:

A relation R is in BCNF with respect to a set F of functional dependencies, if for all the functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

$\alpha \rightarrow \beta$ is a trivial functional dependency, i.e., $\beta \subseteq \alpha$ (β is contained in α)

α is a super key for schema R

In other words, every determinant in a non-trivial dependency is a (super) key.

Let us consider some examples of relations that are in the BCNF or not. Consider the following customer relation:

CUSTOMER (Cid, Cname, Caddress)

The CUSTOMER relation in the BCNF, as the Cid → Cname, Caddress and Cid is the candidate key.

Now, consider a COURSE relation, as shown in *Table 5.12*:

Student	Course	Teacher

Student	Course	Teacher
Ram	Database	Sham
Rahim	Database	Sham
Reema	Database	Shri
Sanjay	Python	Javed
Vaidehi	Python	Sarita
Aman	Python	Javed

Table 5.12: *COURSE (Student, Couse, Teacher)*

Here, the Course relation is as follows:

$$\{\text{Student} + \text{Course}\} \rightarrow \text{Teacher}$$

It satisfies the rules of database design and normalization. The course relation is in the 3NF, but not in BCNF. The reason for FD $\{\text{Teacher}\} \rightarrow \{\text{Course}\}$ is non-trivial and $\{\text{Teacher}\}$ does not contain a candidate key. So, to satisfy the requirement of BCNF, we can decompose the Course relation. After the decomposition, the relations will be as follows:

$$\text{Student} \rightarrow \text{Course}$$

and

$$\text{Teacher} \rightarrow \text{Course}$$

Now, the relations are in the BCNF, but we have lost the FD $\{\text{Student, Course}\} \rightarrow \{\text{Teacher}\}$. So, it is not always possible to satisfy the design goals as the conversion of relation into BCNF, lossless join, and dependency preservation. Here, lossless means the data should not be lost or created when decomposing the relations. When compared with the normalization to 3NF, it is always lossless and dependency preserving, whereas the normalization to BCNF is lossless, but may not preserve all the dependencies as we have seen in the course relation.

Fourth Normal Form (4NF)

Multivalued dependency occurs when there are more than one independent multivalued attributes in a table. Let us consider the example of Sachin Tendulkar related to the dependents and the incomes sources, as shown in *Table 5.13*; we want to store this information in the database as PERSON relation:

PName	Dependent	Income_Source
Sachin Tendulkar	Arjun Tendulkar	Cricket
Sachin Tendulkar	Sara Tendulkar	Advertisement
Sachin Tendulkar	Anjali Tendulkar	Hotel

Table 5.13: *Sachin Tendulkar Dependents and Incomes sources*

A tuple in PERSON relation represents the fact that Sachin Tendulkar can earn from three different income sources and has three dependents. This indicates that a person can earn from different income sources and have more than one dependent. The important thing is that the income source and the dependents are not directly related to one another. To keep these types of tuples in the relation consistent, we must keep a tuple to represent every combination of a person's dependent and an income source. This constraint is specified as a multi-valued dependency on the Person relation. Informally, whenever two independent 1:N relationships, A:B and A:C, are mixed on the same relation, and Multi Valued Dependency (MVD) may arise. In fact, the multi-valued dependencies are a consequence of the first normal form. The first normal form does not allow repeating groups or an attribute in a tuple to have a set of values. If we have two or more multivalued independent attributes in the same relation schema, we get into the problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation instance consistent. This constraint is specified by a multi-valued dependency.

Let us formally define the multi-valued dependency (MVD i.e $A \twoheadrightarrow B$) and the fourth normal form, in the following section.

Multi-valued functional dependency

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$ be the multi-valued dependency $\alpha > \beta$ that holds on Ri if in any legal relation r(R), for all pairs of tuples t1 and t2 in r such that, the following takes place:

$$\begin{aligned} t1[\alpha] &= t2[\alpha] = t3[\alpha] = t4[\alpha] \\ t3[\beta] &= t1[\beta] \\ t3[R-\beta] &= t2[R-\beta] \\ t4[\beta] &= t2[\beta] \\ t4[R-\beta] &= t1[R-\beta] \end{aligned}$$

So, the multi-valued functional dependency $\alpha > \beta$ is said to be a trivial functional dependency if $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$.

The formal definition of multi-valued functional dependency specifies that, given a particular value of PName attribute, the set of values of the dependent attribute determined by this value of PName is completely determined by Pname alone and does not depend on the values of the remaining attributes Income_Source of the relation schema PERSON relation. The PERSON schema is in BCNF because no functional dependencies hold in PERSON. Therefore, we need to define a fourth normal form that is stronger than BCNF and which disallows the relation schemas, such as PERSON.

A relation schema R is in the fourth normal form with respect to a set D of functional and multi-valued dependencies, if for all the multi-valued dependencies in D+ of the form $\alpha > \beta$ where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

$\alpha > \beta$ is a trivial multivalued dependency

It should be in BCNF

α is a super key for schema R.

The definition of 4NF differs from BCNF only in the use of multi-valued dependency instead of the use of functional dependency.

Example:

CName > CStreet, CCity

This means that the customer may have more than one legal address for a bank account or loan account number, as shown in *Table 5.14*:

LAnumber	CName	CStreet	CCity
LA330	Amol	LBS	Mumbai
LA330	Amol	MGR	Mumbai
LA331	Ansh	LBS	Pune
LA900	Vaidehi	LBS	Pune
LA330	Amol	LBS	Pune

Table 5.14: *Customer Loan Relation*

Every functional dependency is also a multivalued functional dependency.

Join dependencies and Fifth Normal Form (5NF) or Project Join Normal Form (PJNF):

Join Dependency

Let R be a relation, and let A, B, and Z be arbitrary subsets of the set of attributes of R. Then, we say that R satisfies the join dependency as follows:

R * (A, B,……Z)

This happens, if and only if R is equal to the join of its projections on A, B, C……..Z.

Example: If we agree to use SP to mean the subset { S#, P#} of a set of attributes of SPJ, and similarly for PJ and JS, then relation SPJ satisfies the JD*(SP, PJ, JS), as shown in *Table 5.15*:

S#	P#	J#
S1	P1	J2
S1	P2	J1
S2	P1	J1

S#	P#	J#
S1	P1	J1

SP		PJ		JS	
S#	P#	P#	J#	J#	S#
S1	P1	P1	J2	J2	S1
S1	P2	P2	J1	J1	S1
S2	P1	P1	J1	J1	S2

Join over J#, S#		
S#	P#	J#
S1	P1	J2
S1	P2	J1
S2	P1	J1
S2	P1	J2
S1	P1	J1

Table 5.15: *SPJ table*

SPJ is a join of all three of its binary projections but not of any two.

A relation is said to be in a 5NF if the candidate keys of R imply every join dependency in R.

Relation SPJ is not in 5NF for the following reasons:

- It can be 3-decomposed.
- That 3-decomposability is not implied by the fact that the combination { S#, P#, J#} is a candidate key of relation.

After the decomposition, the three projections SP, PJ, and JS are each in 5NF, since they do not involve any JDS at all.

In real-life applications, any task requires more than one operation and must be completed as a single task. The task is completed only when all the operations are completed successfully. This group of operations forms a single logical unit of work called transactions which is of utmost importance in the theory of database systems. So, we will discuss transactions and related concepts in the next chapter.

Conclusion

Normalization is a very important step in database design. Functional dependencies help maintain the quality of database design. It is a relationship that exists between two attributes.

The functional dependency of Fid on FName is represented by Fid → FName.

Using the normalization process, we can reduce data redundancy database anomalies. It also helps improve data integrity. Normalization has various levels, such as 1NF, 2NF, 3NF, BCNF, 4NF, and 5NF.

If we have not normalized the database in any normal form, it is in UNF.

1NF disallows composite attributes, multivalued attributes, and nested relations, i.e., attributes whose values for an individual tuple are non-atomic.

A relation schema R is in 2NF if the relation is in 1NF and if every non-prime attribute A in R is fully functionally dependent on the primary key. A relation schema R is in 3NF if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key.

A relationship is said to be in BCNF if it is already in 3NF and the left-hand side of every dependency is a candidate key. A relation is said to be in 4NF if each table contains not more than one multivalued dependency per key attribute.

A relation is in 5NF if it is in 4NF and does not contain any join dependency and joining should be lossless.

In the next chapter, we will study transaction management. In transaction management, you will get familiarized with the basics of the transaction, ACID properties, and the different states of a transaction. We will also learn about concurrency control, deadlock and the various recovery techniques.

Questions

1. Considering a schema R (A, B, C, D) and functional dependencies A -> B and C -> D, which of the following is the decomposition of R into R1 (A, B) and R2(C, D)?
    (a) Dependency preserving and lossless join
    (b) Lossless join but not dependency preserving
    (c) Dependency preserving but not lossless join
    (d) Not dependency preserving and not lossless join
2. Given a relation R (A, B, C, D) and Functional Dependency set FD = { AB → CD, B → C }, determine whether the given R is in 2NF. If not, convert it into 2NF.
3. Which of the following is TRUE?
    (a) Every relation in 2NF is also in BCNF.
    (b) A relation R is in 3NF if every non-prime attribute of R is fully functionally dependent on every key of R.

(c) Every relation in BCNF is also in 3NF.

(d) No relation can be in both BCNF and 3NF.

4. What is the purpose of normalization in DBMS?
5. Explain 1NF and 2NF with an example.
6. What are the different types of normalization? Explain 3NF and BCNF with an example.
7. Explain the term Functional Dependency.
8. What is normalization?
9. Define normalization. Discuss the different normalization Techniques with examples.
10. Explain the different normal forms.

CHAPTER 6

Transactions Management and Concurrency and Recovery

Introduction

A transaction is a logical unit of work that represents a real-world entity, whereas concurrency control is the management of concurrent transaction execution. Concurrency control deals with interleaved execution of more than one transaction. In this chapter, you will learn about concurrency control and the various recovery techniques. You will also learn what is serializability and how to find whether a schedule is serializable or not.

Structure

In this chapter, we will cover the following topics:

- Transaction and ACID properties
- Concurrency control and its importance in maintaining the database's integrity
- Concurrency control and various locking methods
- Recovery techniques to recover from crashes
- Deadlock prevention and detection techniques

Objective

After studying this chapter, you will get familiarized with the basics of transaction and ACID properties. You will learn how to apply transaction processing mechanisms in relational databases. You will then learn the principles of concurrency control and recovery management. You will also learn how to avoid, prevent, or detect the deadlock in advance.

Transaction

The dictionary meaning of transaction is a piece of business that is performed between people. The transaction concept might have been derived from contract law. Of course, a contract is simply an agreement. In making a contract, two or more parties negotiate for a while and then make a deal. The deal is bound by the joint signature on a document or by some other act (as simple as a handshake or signal). Translating the transaction concept from contract law to the realm of computer science, we observe that most of the transactions we see around us (banking, ticket reservation, engineering admission process for seat reservation, a car rental system, or buying groceries) may be reflected in a computer as transformations of a system state. So, we can simply say that a transaction is an operation or a set of operations that are all logically related and act as a single logical unit of work. It can be performed by a database user or application program to read or update the contents of the database.

For example, in the case of the railway reservation system, it will be a fair deal between the person who would like to reserve the ticket and the railway reservation system. Here, a system state consists of the records and devices with changeable values. The system state includes the assertions about the values of records and the allowed transformations of the values. These assertions are called system consistency constraints. The system provides actions that read and transform the values of records and devices. A collection of actions that comprise a consistent transformation of the state may be grouped to form a transaction. Transactions preserve the system consistency constraints – they obey the laws by transforming the consistent states into new consistent states. Systems that use transactions must guarantee the basic transaction properties, namely, atomicity, consistency, isolation, and durability (also called the ACID properties).

In this chapter, we will discuss these ACID properties with the movie ticket booking system.

Nowadays, movie tickets are mostly booked online through the movie ticket booking systems such as BookMyShow. To book the movie tickets without any error, the ticket booking system must preserve the ACID properties. Transactions must be atomic and durable, i.e., either all the actions are done and the transaction is said to commit, or none of the effects of the transaction survive and the transaction is said to abort. These definitions need slight refinement to allow some actions to be ignored and to account for the others that cannot be undone.

The actions on the entities are categorized as follows:

- **Unprotected**: The action need not be undone or redone if the transaction must be aborted or the entity value needs to be reconstructed.
- **Protected**: The action can and must be undone or redone if the transaction must be aborted or if the entity value needs to be reconstructed.
- **Real**: Once done, the action cannot be undone.

The operations on temporary files and the transmission of intermediate messages are examples of unprotected actions. Conventional database and message operations are examples of protected actions. Transaction commitment and operations on real devices, such as cash dispensers from ATMs, are examples of real actions.

Each transaction is defined as having exactly one of the two outcomes – committed or aborted. All protected and real actions of committed transactions persist, even in the presence of failures. On the other hand, none of the effects of protected and real actions of an aborted transaction are ever visible to the other transactions.

Once a transaction commits, its effects can only be altered by running further transactions. For example, if someone is underpaid, the corrective action is to run another transaction that pays an additional sum. Adjustment of a bad transaction is done via further compensating the transactions.

Such post facto transactions are called compensating transactions. The transaction concept emerges with the following properties:

- **Atomicity**: It either happens or it does not; either all are bound by the contract or none are.
- **Consistency**: The transaction must obey legal protocols.
- **Isolation**: No interference from concurrently executing transactions.

- **Durability**: Once a transaction is committed, it cannot be abrogated.
- **Serializability**: Important in a multiuser database and for the concurrent execution of the transactions.

Transactions can be simple or complex. A simple transaction consists of a linear sequence of operations. A complex transaction is the one where one operation triggers another operation or initiates a transaction depending on the outcome of the previous operations.

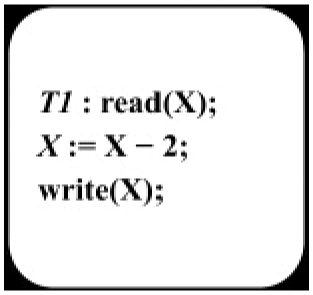
When only one transaction is executed at any instance of time in a database system, serializability and isolation of the database are automatically maintained. This is observed in a single-user database system. In a multi-user database system, since multiple transactions execute concurrently, serializability and isolation must be guaranteed by the implementation of controls. Yet the other ACID properties such as atomicity, consistency, and durability need to be ensured by the database system, be it a single-user or a multi-user. A multi-user database system is specifically required to implement the concurrency control techniques to preserve the ACID properties. Apart from the ACID properties, a database system must manage recovery from the application errors as well as errors introduced by the operating system interruptions and power failure.

To better understand the transaction concept, let us continue with the example of a movie ticket booking system. A movie ticket booking system consists of multiple theatres, multiple screens, various movies, different movie slots, category-wise seating capacity, pricing, and numerous customers. In order to book a movie ticket for a customer on a particular screen of a theatre for a definite movie slot, a movie ticket booking system consists of a set of transactions. In some cases, after the ticket booking, the customer might be willing to change the ticket to another movie slot rather than cancelling the ticket. In a relational database for the movie ticket booking systems, we refer to these terms as database items for the ease of understanding. These database items can be read or written by a transaction.

Therefore, for the sake of simplicity, let us start with the two main operations in a database transaction – read operation and write operation – as described as follows:

- Read operation is used to read the data from the database. For example, for us, it refers to a ticket data item and checks the availability from the database.
- Write operation is used to insert or update the data value in the database. For example, Write(X) means to write the updated value of X into the database. For us, it stores the number of tickets available after booking the tickets for a customer.

It is important to know whether a change to a data item appears only in the main memory or if it has been written to the database on the disk. In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored elsewhere and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately. Let T1 be a transaction that books two tickets; this transaction can be defined as shown in *Figure 6.1*:



```
T1 : read(X);
X := X − 2;
write(X);
```

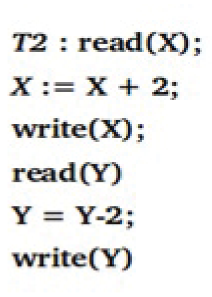
Figure 6.1: *Transaction T1*

Let us now discuss the transaction properties, i.e., the ACID properties for the movie ticket booking system, as follows:

- **Consistency**: The consistency requirement in the ticket booking system is that the total number of tickets will always remain intact. This means, the sum of available and reserved tickets will always be the maximum number of tickets for a particular show. For simplicity, consider the maximum number of tickets that is available is 100. In our case, the sum of available and booked tickets should always be 100. Without the consistency requirement, the tickets could be created or destroyed by the transaction. It can be verified easily that, if the database is consistent before the execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of the integrity constraints.

- **Atomicity**: To elaborate on the Atomicity, let us consider that the customer has booked the ticket for Saturday, 24th July 2021, but would now like to change it to Sunday, 25th July 2021. Consider that for 25th July 2021, just before the execution of transaction T2, all (100 tickets) tickets were available for booking. So, he will update the booking of the 24th July 2021 to 25th July 2021 as the facility is available, as shown in *Figure 6.2*:



```
T2 : read(X);
X := X + 2;
write(X);
read(Y)
Y = Y-2;
write(Y)
```

Figure 6.2: *Transaction T2 updating 24th July ticket to 25th July 2021*

Now, suppose that during the execution of transaction T2, a failure occurs that prevents transaction T2 from completing its execution successfully. Further, suppose that the failure happened after the write(X) operation but before the write(Y) operation. In this case, the number of tickets for 24th July and 25th July will remain intact, as internally the tickets for 24th July have been canceled, and before updating the customer's ticket for 25th July, the transaction failed. But from the customer's point of view, the transaction is in inconsistent state. In this case, the adjustment of a bad transaction or failed transaction may be done via a compensating transaction. This calls for the need for atomicity property. An atomic property ensures that either all operations of a transaction are completed in the database or none. To implement atomicity, for the data items on which a transaction performs the write operation, the database system maintains and tracks the old values in a log file. In case the transaction is not completed successfully, the old values from the log files are restored in the database system, thus ensuring the atomicity of the transaction. A recovery system specifically serves this purpose in a database system.

- **Durability**: It ensures that once the execution of a transaction completes successfully, the changes done on the database persist, even if there is a system failure. Here, the assumption is that in a computer system, we may lose the data in the main memory, but the data written to the disk always persists. In addition, if the transactions are implemented with logging, then the transaction manager may be used to reconstruct the system state from an old state plus the log. This provides transaction durability in the presence of multiple failures, that is, we can guarantee durability by ensuring updates of the transactions on the disk, and this is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure. The recovery system of the database ensures the durability and atomicity properties of the transaction.
- **Isolation**: In a multi-user database, the concurrency of the system plays a major role and has many advantages. Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently in an uncontrolled manner, their operations may interleave in some undesirable way, resulting in an inconsistent state. The Isolation property of a transaction ensures that the data used during the execution of the first transaction cannot be used by any other transaction until the first one is completed.

The database is temporarily inconsistent while the transaction updates the data items. We can illustrate some of these problems by referring to a simplified BookMyShow example.

For example, in the case of BookMyShow, any number of customers can reserve the tickets. Let us consider a scenario of only the last five seats remaining and two customers are trying to reserve the seats. Now, during the execution, the database is temporarily inconsistent while the transaction assigns the tickets to the customers. In this case, both the customers, i.e., the current transactions may read the intermediate inconsistent value, which is, five tickets were available. If the first customer reserves three tickets and the second reserves four tickets, then the database may be left in an inconsistent state as it produces spurious tickets, even after both the transactions have been completed. To avoid the problem of concurrently executing the transactions, the generation of spurious tickets is handled by the isolation property of the transaction. Isolation is the database-level property. It controls how and when the changes are to be made to the database. The main goal of the isolation property is the concurrent execution of the transactions without adversely affecting the execution of each. So, ensure that the isolation property is the responsibility of the concurrency-control system of the database system.

In order to maintain consistency, accuracy, completeness, and data integrity in a database, there is a need to understand the Transaction State Diagram, as a transaction can fail during execution. So, if we make the changes in the actual database instead of the local memory, the database may be left in an inconsistent state in case of any failure. To make it clear, let us understand the Transaction State Diagram.

Transaction State Diagram

In DBMS, the transaction goes into many states throughout its lifetime. At a time it can be in one of the following states, as shown in *Figure 6.3*:

- Active state
- Partially committed state
- Committed state
- Failed state
- Aborted state
- begin Terminated state

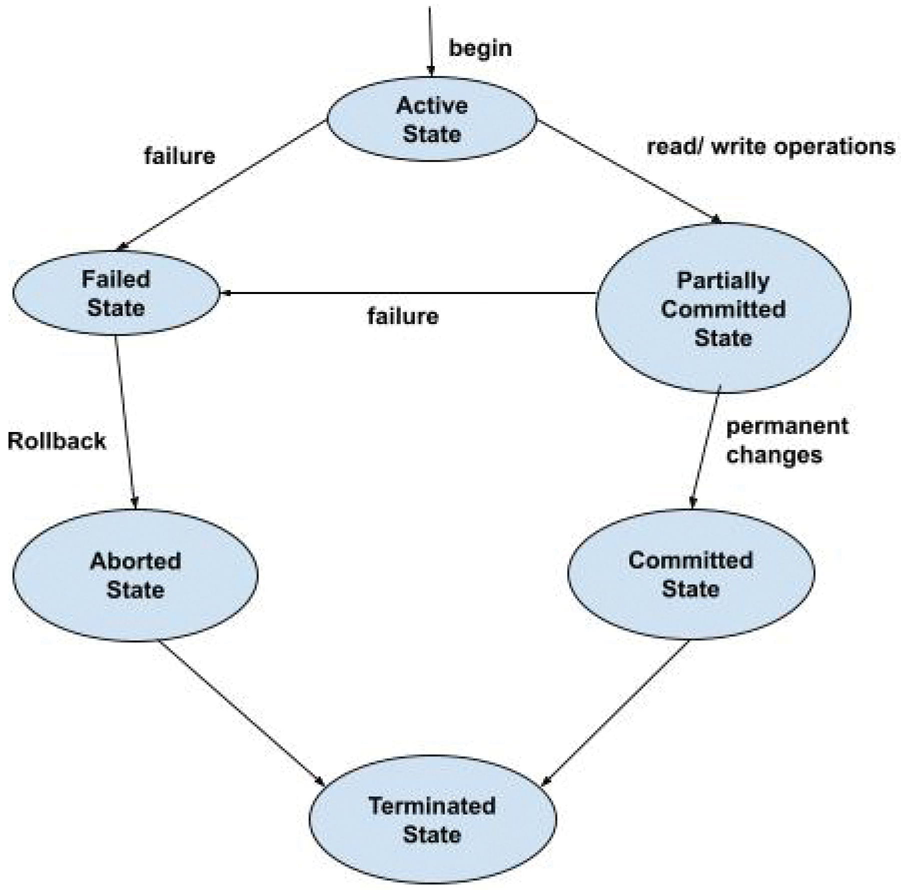


Figure 6.3: *Transaction State Diagram*

The description of these transaction states are as follows:

- **Active state**: This is the first state of every transaction. When all the read and write operations of the transaction are executed, it remains in the "*Active State*". If all the operations are performed without any error, it goes to the "*Partially Committed State*"; if any failure occurs, it goes to the "*Failed State*".
- **Partially committed state**: When the transaction executes its last operation, it goes into a partially committed state where the changes are made in the main memory instead of the actual database. If the changes are made permanent in the database, it will enter into the "*Committed State*". If a failure occurs before making the changes permanent in the Database, it will go to the "*Failed State*".
- **Committed state**: A transaction goes from the partially committed state to the committed state when a transaction executes all its

operations successfully, followed by all the changes made in the local memory during the execution of the transaction is permanently stored in the database. Here, our assumption is that the database resides on the hard disk or secondary storage and the data stored on it persists even after failure.

- **Failed state**: If a transaction is being executed and either a hardware failure or a software failure occurs, the transaction goes into "*Failed State*" from the active state. In case a transaction is in a partially committed state and a failure occurs, even then it goes into the "*Failed State*". As in the case of a partially committed state, the changes made have not yet been reflected on the persistent storage.
- **Aborted state**: If a transaction fails during execution, it goes into a "*Failed state*". At that time, the changes made into the local memory are supposed to be rolled back to their previous consistent state and the transaction goes into the "*Aborted State*" from the failed state. At this stage, the system has two options – restart the transaction or kill the transaction.

Serializability – Conflict and View

As we discussed, concurrent transactions provide less response time and high throughput. At the same time, it may lead the database into an inconsistent state and the ACID properties of the transaction cannot be held.

The term serializability will help solve the preceding problem. Serializability ensures that a given transaction schedule will never lead to an inconsistent state. When the transactions are executed serially, i.e., one after the other, it always achieves a consistent state. So, we can say a serial schedule is always serializable.

However, when we execute multiple transactions concurrently, the schedule is called a non-serial schedule. This non-serial schedule should be serializable to maintain consistency.

In short, serializability means generating a serial schedule for the concurrent transactions in such a way that it will always achieve consistency. The different types of Serializability are shown in *Figure 6.4*:

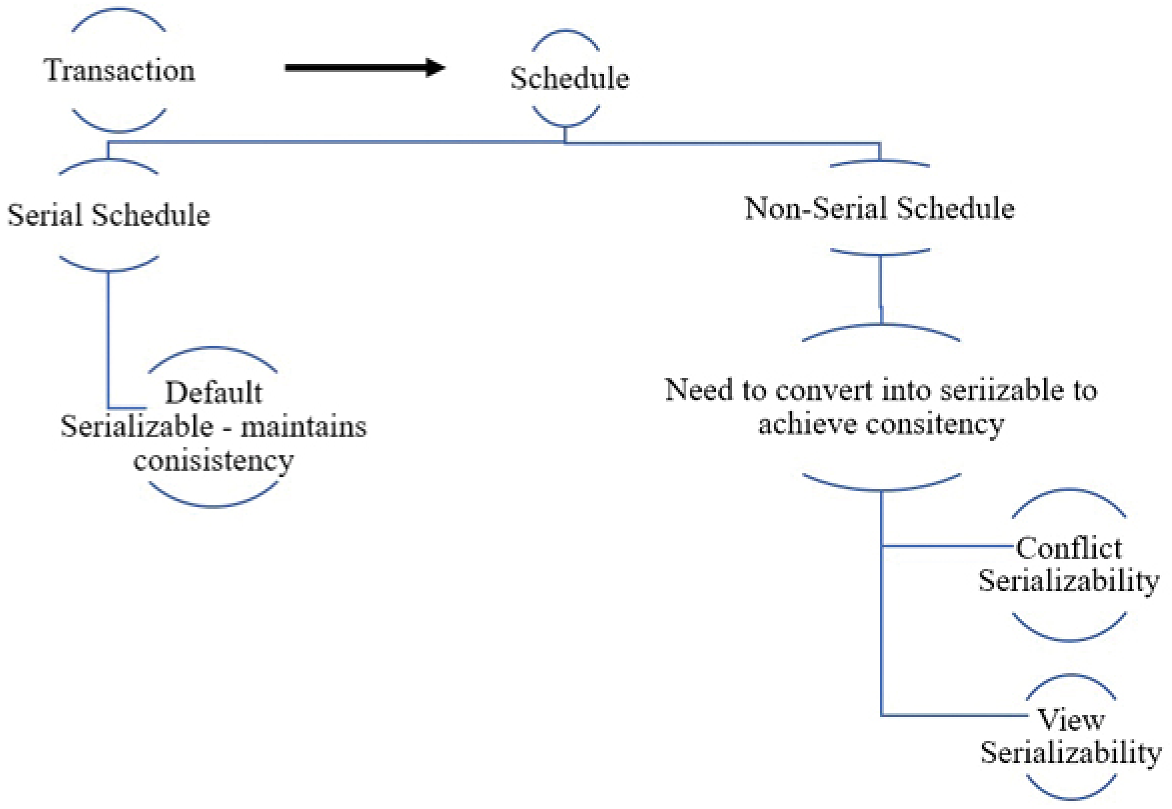

Transaction
Schedule
Serial Schedule
Non-Serial Schedule
Default Serializable - maintains conisistency
Need to convert into seriizable to achieve consitency
Conflict Serializability
View Serializability

Figure 6.4: *Types of Serializability*

Conflict Serializability: Its emphasis is on swapping the non-conflicting operations among two different transactions. After this swap, the generated schedule is known as conflict serializability.

Let us understand the conflicting operations in more detail.

Two operations are said to conflict if they satisfy all the following three conditions:

- The two operations are part of two different transactions, T1 and T2.
- The two operations are using the same data item, let's say X.
- Among these two operations, at least one of the operations is a write operation.

With the help of the ticket reservation system, let us understand it in more detail. In any ticket reservation system, when a user tries to book a ticket, first, he sees the availability of the seats at that moment in the chosen location. For example, in the movie ticket reservation system, the user first checks the available tickets for the particular movie in his chosen theater. Similarly, in the airway/railway ticket reservations, the users see the available tickets for a chosen trip on a specific date.

This check operation is treated as a read operation of the database. If the two users check the available seats for a movie or available seats for airway/railway, these operations are non-conflicting as they are only read operations.

Now, let us consider the scenario where two users, U1 and U2, see the number of available seats. They try to book the seat at the same time. The booking of the seat is considered as a write operation. When the two users try to perform a write operation simultaneously on the same variable, it leads to inconsistency.

Conflict serializability focuses on generating a schedule by swapping the non-conflicting operations among the transactions.

Assume in the movie ticket reservation system, U1 and U2 are booking the ticket at the same time. Each user's action is considered as a separate transaction, T1 and T2.

Case 1: U1 and U2 are booking tickets serially (U2 is followed by U1), as shown as follows:

T1(U1:Booking a movie ticket)	T2(U2: Booking a movie ticket)

Read_available_Seats(X)
Available_Seats(X)=
available_seats(X)-book_seats
Write_available_Seats(X)
	Read_available_Seats(X)
	Available_Seats(X)=
	available_seats(X)-book_seats
	Write_available_Seats(X)

Case 2: U1 and U2 are booking tickets serially (U1 is followed by U2), as shown as follows:

T1(U1:Booking a movie ticket)	T2(U2: Booking a movie ticket)

T1(U1:Booking a movie ticket)	T2(U2: Booking a movie ticket)
	Read_available_Seats(X)
	Available_Seats(X)= available_seats(X)-book_seats
	Write_available_Seats(X)
Read_available_Seats(X)	
Available_Seats(X)= available_seats(X)-book_seats	
Write_available_Seats(X)	

Both Case 1 and Case 2 are examples of a serial schedule. They are serializable by nature.

Case 3: Both users, U1 and U2, are booking the movie ticket at the same time, as shown as follows:

T1(U1:Booking a movie ticket)	T2(U2: Booking a movie ticket)
Read_available_Seats(X)	Read_available_Seats(X)
Available_Seats(X)= available_seats(X)- book_seats	Available_Seats(X)= available_seats(X)-book_seats
Write_available_Seats(X)	
	Write_available_Seats(X)

OR

T1(U1:Booking a movie ticket)	T2(U2: Booking a movie ticket)
Read_available_Seats(X)	
Available_Seats(X)= available_seats(X)- book_seats	Read_available_Seats(X)
	Available_Seats(X)= available_seats(X)-book_seats
Write_available_Seats(X)	Write_available_Seats(X)

In both these cases, whether U1 or U2 click first on the ‘book the tickets’ button, it will lead to inconsistency.

If we draw a directed graph for the preceding schedule, it will show a cycle in it, as shown in *Figure 6.5*:

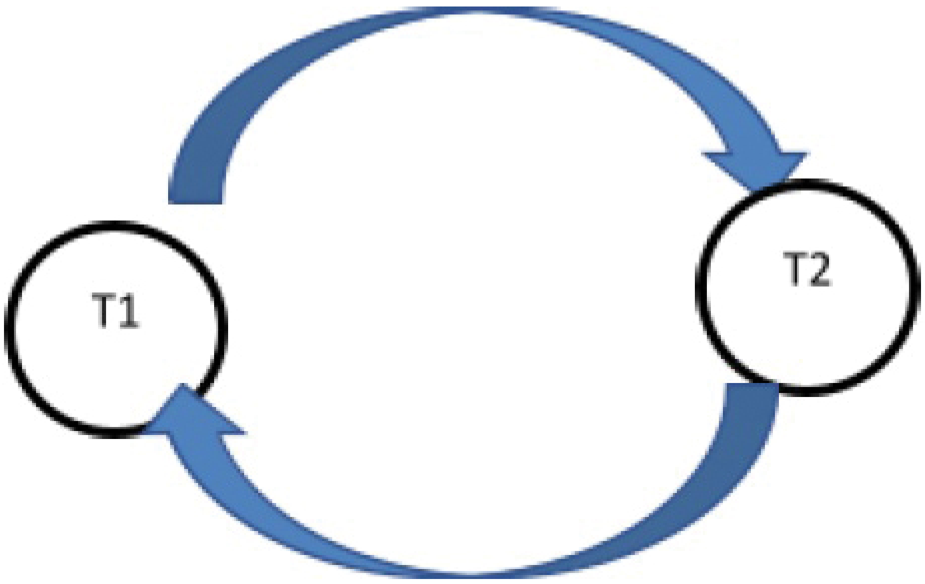


Figure 6.5: *Directed graph of transaction T1 and T2 (with cycle)*

Now, consider another scenario where U1 and U2 are booking the tickets simultaneously for two movies.

Case 1: U1 and U2 are booking tickets serially (U2 is followed by U1), as shown as follows:

T1(U1:Booking a movie ticket for two movies)	T2(U2: Booking a movie ticket)

T1(U1:Booking a movie ticket for two movies)	**T2(U2: Booking a movie ticket)**
Read_available_Seats_Movie1(X)	
Available_Seats_Movie1(X)= available_seats_Movie1(X)-book_seats_Movie1	Read_available_Seats_Movie1(X)
Write_available_Seats_Movie1(X)	Available_Seats_Movie1(X)= available_seats_Movie1(X)-book_seats_Movie1
	Write_available_Seats_Movie1(X)
Read_available_Seats_Movie1(Y)	
Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1	Read_available_Seats_Movie1(Y)
Write_available_Seats_Movie1(Y)	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1
	Write_available_Seats_Movie1(Y)

Case 2: U1 and U2 are booking tickets serially (U1 is followed by U2), as shown as follows:

T1(U1:Booking a movie ticket for two movies)	**T2(U2: Booking a movie ticket)**

T1(U1:Booking a movie ticket for two movies)	T2(U2: Booking a movie ticket)
Read_available_Seats_Movie1(X) Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1 Write_available_Seats_Movie1(X) Read_available_Seats_Movie1(Y) Available_Seats_Movie1(Y)= available_seats_Movie1(Y)- book_seats_Movie1 Write_available_Seats_Movie1(Y)	Read_available_Seats_Movie1(X) Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1 Write_available_Seats_Movie1(X) Read_available_Seats_Movie1(Y) Available_Seats_Movie1(Y)= available_seats_Movie1(Y)- book_seats_Movie1 Write_available_Seats_Movie1(Y)

Again here, both Case 1 and Case 2 are examples of serial schedules. They are serializable by nature.

Consider Case 3 where both the users, U1 and U2, book the movie ticket at the same time.

Case 3: U1 and U2 are booking the seats for two movie tickets at the same time, as shown as follows:

T1(U1:Booking a movie ticket for two movies)	T2(U2: Booking a movie ticket)
Read_available_Seats_Movie1(X) Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1	
	Read_available_Seats_Movie1(X) Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1
Write_available_Seats_Movie1(X)	

Read_available_Seats_Movie1(Y)	

	Write_available_Seats_Movie1(X)
	Read_available_Seats_Movie1(Y)

Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1 Write_available_Seats_Movie1(Y)	
	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1 Write_available_Seats_Movie1(Y)

Figure 6.6 shows the directed graph for the preceding schedule, and it shows a cycle in it:

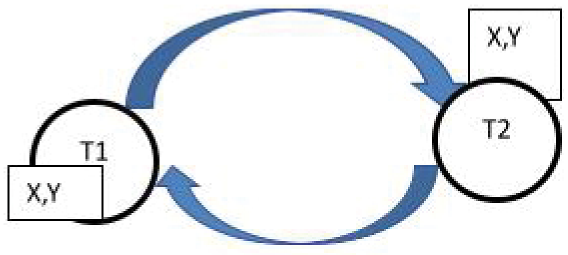


Figure 6.6: *Directed graph of transaction T1 and T2 (for Movie Ticket example)*

Case 4: Swap only the non-conflicting operations, as shown as follows:

T1(U1:Booking a movie ticket for two movies)	T2(U2: Booking a movie ticket)

T1(U1:Booking a movie ticket for two movies)	T2(U2: Booking a movie ticket)
Read_available_Seats_Movie1(X)	
Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1	Read_available_Seats_Movie1(X)
Write_available_Seats_Movie1(X)	Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1
	Write_available_Seats_Movie1(X)
Read_available_Seats_Movie1(Y)	
Available_Seats_Movie1(Y)= available_seats_Movie1(Y)- book_seats_Movie1	Read_available_Seats_Movie1(Y)
Write_available_Seats_Movie1(Y)	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)- book_seats_Movie1
	Write_available_Seats_Movie1(Y)

The directed precedence graph for the preceding schedule is shown in *Figure 6.7*; it does not consist of a cycle that indicates that the schedule is conflict serializable:

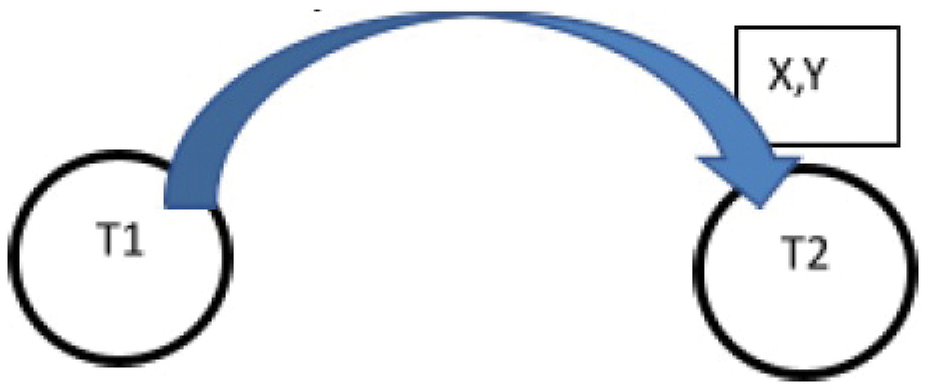


Figure 6.7: *Directed graph of transaction T1 and T2*

View serializability: This is an alternative way to check a given schedule of the non-serial transactions that can be serialized. In view serializability, the main task is to cross-check, given that a non-serial schedule is view equivalent to its serial schedule.

Let us understand the concept with a generic example. Consider a serial schedule of transactions T1 and T2, each consisting of four operations.

Case 1: S1-A serial schedule T1 is followed by T2, as shown as follows:

T1	T2
R(X)	
W(X)	
R(Y)	
W(Y)	R(X)W(X)
	R(Y)W(Y)

Now, consider a non-serial schedule of the same transaction, T1 and T2.

Case 2: S2-A non-serial schedule of T1 and T2 (concurrency with interleave processing using single CPU), as shown as follows:

T1	T2
R(X)	
W(X)	
	R(X) W(X)
R(Y)	
W(Y)	R(Y) W(Y)

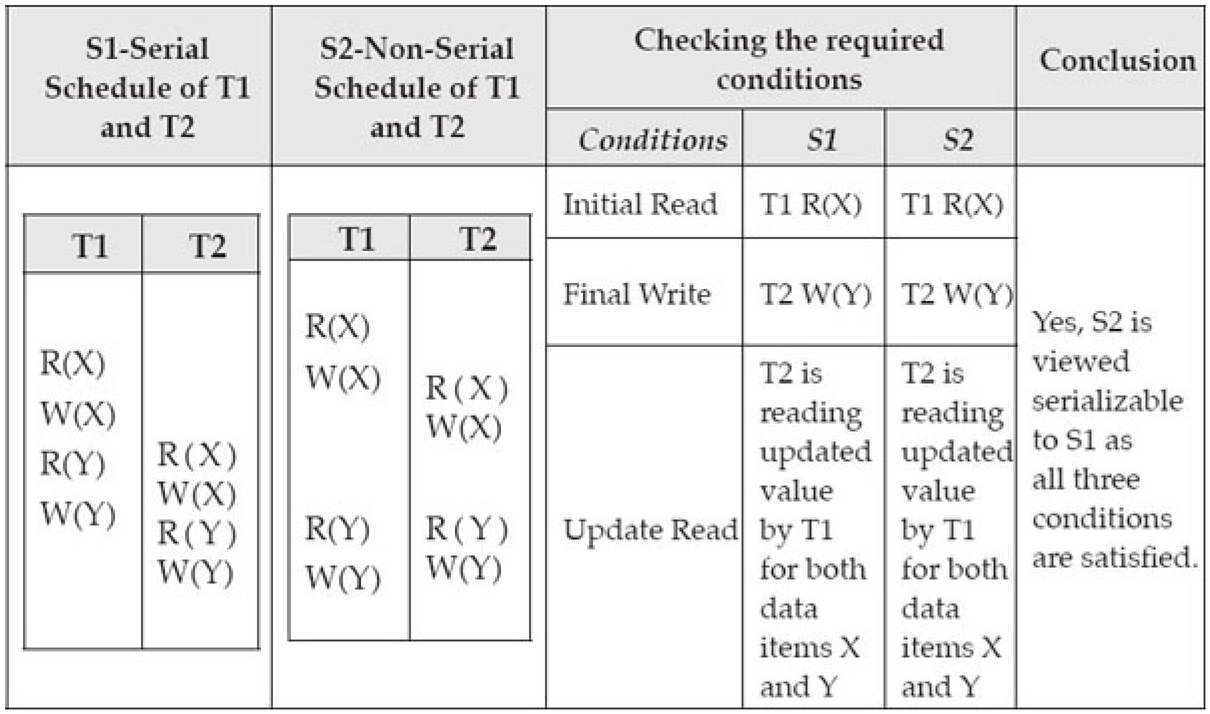
Here, we have to find if the given non-serial schedule is view equivalent to its serial schedule; then it can be called **V**iew Serializable and the process can be termed as View Serializability.

The non-serial schedule of the given transactions can be called as view serializable to its serial schedule if it satisfies the following conditions:

- **Initial read**: Initial read in each schedule must match. For example, in the S1 schedule, if the transaction T1 is reading data item X before transaction T2, then in the S2 schedule too, transaction T1 should read data item X before transaction T2.

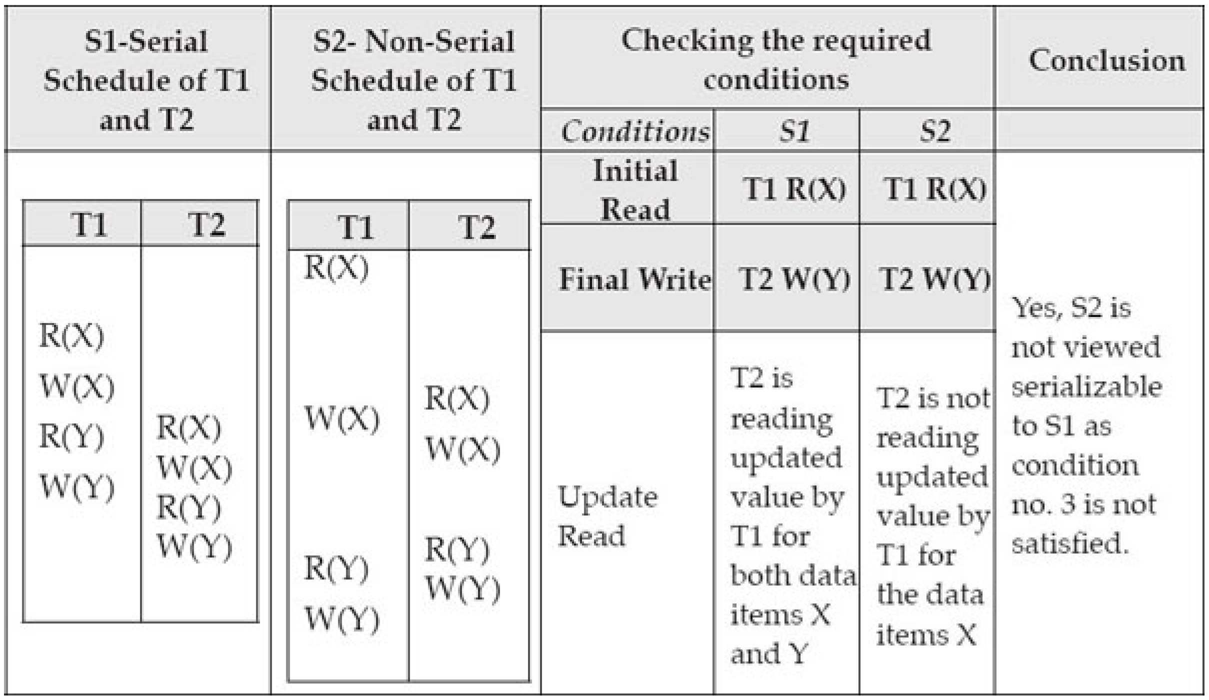
- **Final write**: Similar to the initial read, the final write in each schedule also must match. For example, in the S1 schedule, if the transaction T1 is performing the final write on data item X, then in the S2 schedule too, transaction T1 should perform the final write on data item X.
- **Update read**: If in schedule S1, the transaction T1 is reading a data item updated by T2, then in schedule S2, T1 should read the value after the write operation of T2 on the same data item. For example, in schedule S1, T1 performs a read operation on X after the write operation on X by T2; then in S2, T1 should read X after T2 performs write on X.

Scenario 1: S1 and S2 are View serializable, as shown as follows:



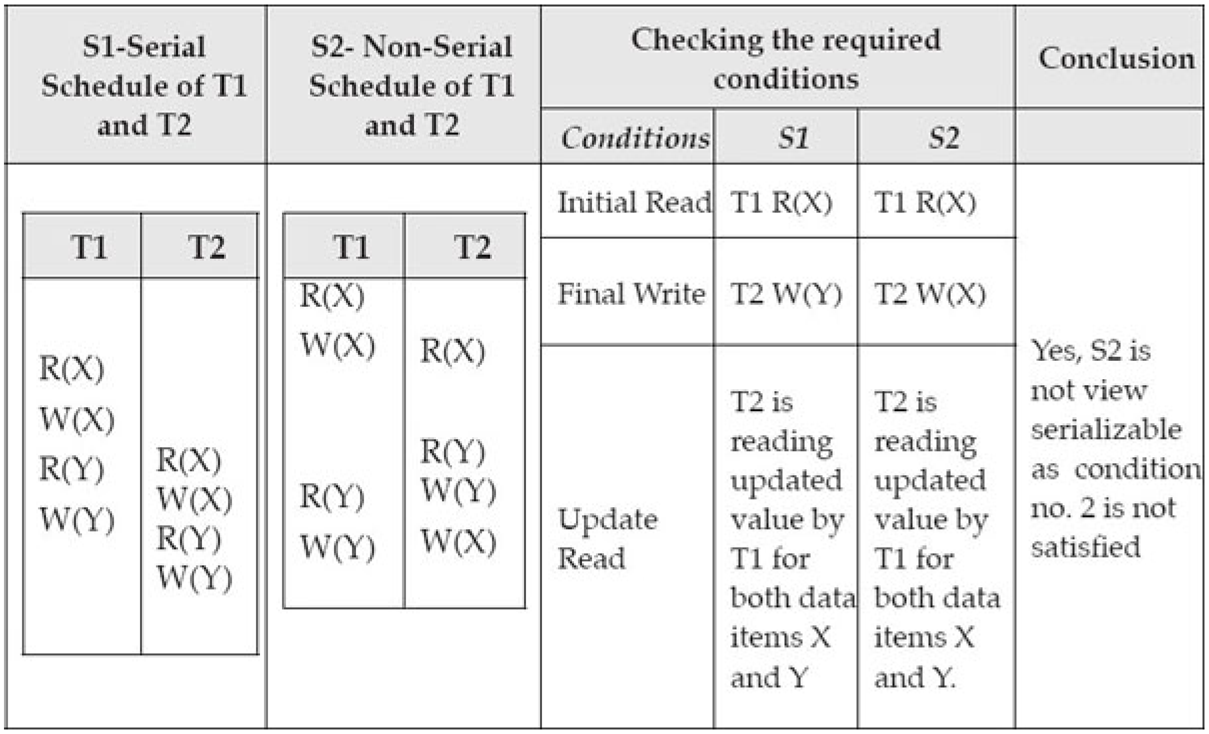
S1-Serial Schedule of T1 and T2	S2-Non-Serial Schedule of T1 and T2	Checking the required conditions			Conclusion
		Conditions	*S1*	*S2*	
T1: R(X) W(X) R(Y) W(Y) T2: R(X) W(X) R(Y) W(Y)	T1: R(X) W(X) T2: R(X) W(X) T1: R(Y) W(Y) T2: R(Y) W(Y)	Initial Read	T1 R(X)	T1 R(X)	Yes, S2 is viewed serializable to S1 as all three conditions are satisfied.
		Final Write	T2 W(Y)	T2 W(Y)	
		Update Read	T2 is reading updated value by T1 for both data items X and Y	T2 is reading updated value by T1 for both data items X and Y	

Scenario 2: S1 and S2 are not view serializable (Condition 3: Update Read is not met), as shown as follows:



S1-Serial Schedule of T1 and T2	S2- Non-Serial Schedule of T1 and T2	Checking the required conditions			Conclusion
		Conditions	*S1*	*S2*	
T1 | T2 R(X) | W(X) | R(Y) | R(X) W(Y) | W(X) | R(Y) | W(Y)	T1 | T2 R(X) | W(X) | R(X) | W(X) R(Y) | R(Y) | W(Y) W(Y) |	Initial Read	T1 R(X)	T1 R(X)	Yes, S2 is not viewed serializable to S1 as condition no. 3 is not satisfied.
		Final Write	T2 W(Y)	T2 W(Y)	
		Update Read	T2 is reading updated value by T1 for both data items X and Y	T2 is not reading updated value by T1 for the data items X	

Scenario 3: S1 and S2 are not View serializable (Condition 2: Final Write is not met), as shown as follows:



S1-Serial Schedule of T1 and T2	S2- Non-Serial Schedule of T1 and T2	Checking the required conditions			Conclusion
		Conditions	*S1*	*S2*	

T1 | T2
R(X) |
W(X) |
R(Y) | R(X)
W(Y) | W(X)
 | R(Y)
 | W(Y)
	T1 | T2
	R(X) |
	W(X) | R(X)
	 | R(Y)
	R(Y) | W(Y)
	W(Y) | W(X)
		Initial Read	T1 R(X)	T1 R(X)	Yes, S2 is not view serializable as condition no. 2 is not satisfied

		Final Write	T2 W(Y)	T2 W(X)	
		Update Read	T2 is reading updated value by T1 for both data items X and Y	T2 is reading updated value by T1 for both data items X and Y.	

Now, try to understand the View serializability with our ticket reservation system.

Scenario 1: S1 and S2 are view serializable, as shown as follows:

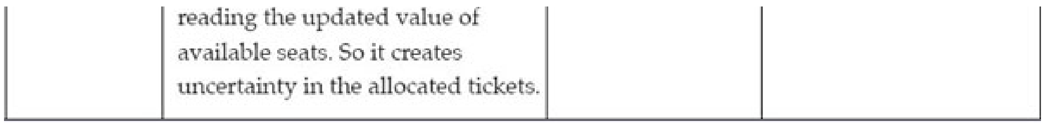
S1-Serial Schedule of T1 and T2		S2- Non-Serial Schedule of T1 and T2	
T1 (U1: Booking a movie ticket for two movies)	T2 (U2: Booking a movie ticket)	T1 (U1:Booking a movie ticket for two movies)	T2 (U2: Booking a movie ticket)
Read_available_Seats_ Movie1(X)	Read_available_Seats_ Movie1(X)	Read_available_Seats_ Movie1(X)	Read_available_Seats_ Movie1(X)
Available_Seats_ Movie1(X)= available_ seats_Movie1(X)-book_ seats_Movie1	Available_Seats_ Movie1(X)= available_ seats_Movie1(X)-book_ seats_Movie1	Available_Seats_Movie1(X)= available_seats_Movie1(X)- book_seats_Movie1	Available_Seats_ Movie1(X)= available_ seats_Movie1(X)- book_ seats_Movie1
Write_available_Seats_ Movie1(X)	Write_available_Seats_ Movie1(X)	Write_available_Seats_ Movie1(X)	Write_available_Seats_ Movie1(X)
Read_available_Seats_ Movie1(Y)	Read_available_Seats_ Movie1(Y)	Read_available_Seats_ Movie1(Y)	Read_available_Seats_ Movie1(Y)
Available_Seats_ Movie1(Y)= available_ seats_Movie1(Y)-book_ seats_Movie1	Available_Seats_ Movie1(Y)= available_ seats_Movie1(Y)-book_ seats_Movie1	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)- book_seats_Movie1	Available_Seats_ Movie1(Y)= available_ seats_Movie1(Y)-book_ seats_Movie1
Write_available_Seats_ Movie1(Y)	Write_available_Seats_ Movie1(Y)	Write_available_Seats_ Movie1(Y)	Write_available_Seats_ Movie1(Y)

Checking the required conditions			Conclusion
Conditions	*S1*	*S2*	
1) Initial Read	Read_available_Seats_Movie1(X)	Read_available_ Seats_Movie1(X)	Yes, S2 is view serializable to S1 as all three conditions are satisfied.
	Interpretation: Both users U1 and U2 are interested in the same movie and check the availability of the seats.		
2) Final Write	Write_available_Seats_Movie1(Y)	Write_available_ Seats_Movie1(Y)	
	Interpretation: Both users U1 and U2 are booking the tickets for the same movie.		
3) Update Read	T2 is reading the updated value by T1 for both data items X and Y	T2 is reading the updated value by T1 for both data items X and Y	
	Interpretation: User U2 is reading the updated value of available seats.		

Scenario 2: S1 and S2 are not View serializable (Condition 3: Update Read is not met), as shown as follows:

S1-Serial Schedule of T1 and T2		S2- Non-Serial Schedule of T1 and T2	
T1 (U1:Booking a movie ticket for two movies)	T2 (U2: Booking a movie ticket)	T1 (U1:Booking a movie ticket for two movies)	T2 (U2: Booking a movie ticket)
Read_available_Seats_Movie1(X)	Read_available_Seats_Movie1(X)	Read_available_Seats_Movie1(X)	Read_available_Seats_Movie1(X)
Available_Seats_Movie1(X)= available_seats_Movie1(X)-book_seats_Movie1	Available_Seats_Movie1(X)= available_seats_Movie1(X)-book_seats_Movie1	Available_Seats_Movie1(X)= available_seats_Movie1(X)-book_seats_Movie1	Available_Seats_Movie1(X)= available_seats_Movie1(X)-book_seats_Movie1
Write_available_Seats_Movie1(X)	Write_available_Seats_Movie1(X)	Write_available_Seats_Movie1(X)	Write_available_Seats_Movie1(X)
Read_available_Seats_Movie1(Y)	Read_available_Seats_Movie1(Y)	Read_available_Seats_Movie1(Y)	Read_available_Seats_Movie1(Y)
Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1	Available_Seats_Movie1(Y)= available_seats_Movie1(Y)-book_seats_Movie1
Write_available_Seats_Movie1(Y)	Write_available_Seats_Movie1(Y)	Write_available_Seats_Movie1(Y)	Write_available_Seats_Movie1(Y)

Checking the required conditions			
Conditions	*S1*	*S2*	
Initial Read	Read_available_Seats_Movie1(X)	Read_available_Seats_Movie1(X)	Yes, S2 is not view serializable to S1 as condition no. 3 is not satisfied
	Interpretation: Both users U1 and U2 are interested in the same movie and check the availability of the seats.		
Final Write	Write_available_Seats_Movie1(Y)	Write_available_Seats_Movie1(Y)	
	Interpretation: Both users U1 and U2 are booking the tickets for the same movie.		
Update Read	T2 is reading the updated value by T1 for both data items X and Y	T2 is not reading the updated value by T1 for both data items X and Y	
	Interpretation: User U2 is not		



	reading the updated value of available seats. So it creates uncertainty in the allocated tickets.		

Concurrent Executions

In a multi-user system, when multiple users access and use the same database at the same time, it is known as concurrent execution of the database. For concurrent execution, we are mainly concerned with the centralized high-performance transaction processing systems and concurrency control methods. Data integrity can be provided at different granularity levels such as table, tuple, attribute, or cell level. Multiple transactions may be executed to update a data item at different granularity levels at the same time. This leads to the need for concurrency control. Concurrency control preserves consistency during the concurrent transaction execution.

It means that many users use the same database simultaneously. The transaction-processing systems usually allow multiple transactions to run concurrently for the following two good reasons:

Improved throughput and resource utilization

Benefits of the CPU-I/O overlap can be gained through transaction concurrency which results in high transaction output by increasing the resource utilization and decreasing the response time. As we know, a transaction consists of many operations and some of these operations are I/O operations and CPU operations, and they can be done in parallel. In other words, the parallelism of the CPU and I/O system can, therefore, be exploited to run multiple transactions in parallel. The obvious result is the concurrent execution of multiple transactions in a given amount of time and this increases the throughput of the system.

Reduced waiting time

The reduced waiting time and increasing efficiency can be justified with the short duration and long duration transactions. Since multiple transactions are under execution, some may be of short duration and some may require long running time. If these mixed transactions run serially, a short transaction may have to wait until the preceding long transaction completes. The adverse effect is unpredictable delays in running a transaction. The best part of a transaction is that if the transactions are operating on different data items of the database, it is better to let them run concurrently. It will share the resources as explained already, i.e., it will share the CPU and I/O devices among them. Thus, the concurrent execution of the transactions reduces the unpredictable delays in the running transactions. Moreover, it also reduces the average response time, i.e., the average time for a transaction to be completed after it has been submitted.

Concurrent execution should be done in an interleaved manner, without affecting the other operations, thus maintaining the consistency of the database. When the concurrent transactions are run in an uncontrolled manner, a few problems may occur which are known as concurrency problems. Let us understand how the concurrent execution of the transactions will result in improved throughput and better resource utilization.

The universally accepted correctness criterion for processing the transactions against a database is serializability, that is, the interleaved execution of a set of concurrent transactions is identical to serial execution. This correctness criterion is not acceptable for some applications such as stock trading bids that may have a first-come, first-served (FCFS) processing requirement.

Let us understand the various problems which may occur during the concurrent executions of the transactions, as follows:

- Lost Update Problems (Write – Write conflict)

  This type of problem occurs when two transactions in the database access the same data item 'A' and have their operations in an interleaved manner that makes the value of data item 'A' inconsistent in the database.

  If there are two transactions, T1 and T2, accessing the same data item 'A' value and then updating it, in such a situation, the second record overwrites the first record, and this problem is known as "*Lost Update problem*", as shown in Table 6.1.

Example: Consider *Table 6.1*, wherein two transactions T1 (credit ₹1000) and T2 (credit ₹2000) perform the credit operation on the same account A where the current balance of account A is ₹2000:

Time	T1	T2	Explanation
t1	Read(A)		At t1 - T1 transaction reads the value of A, i.e., ₹2000
t2	A=A+1000		At t2 - T1 transaction credits in A by ₹1000
t3		Read(A)	At t3 - T2 transaction reads the value of A, i.e., ₹2000
t4		A=A+2000	At t4 – T2 transaction credits in A by ₹2000
t5	Write(A)		At t5 – T1 transaction writes the value of A data item on the basis of value seen at time t2, i.e., ₹3000
t6		Write(A)	At t6 - T2 transaction writes the value of A based on value seen at time t4, i.e., ₹4000 So at time t6, the update of Transaction T1 is lost because Transaction T2 overwrites the value of A without looking at its current value. This type of problem is known as the Lost Update Problem.

Table 6.1: *Lost Update Problem Example*

- Dirty Read Problem (Write-Read conflict)

The dirty read problem occurs when one transaction updates an item of the database, and due to some reason, the transaction fails, and before the data gets rolled back, the updated item's value is accessed by some other transaction, which is the read-write conflict between both the transactions.

Example: Consider *Table 6.2*, wherein two transactions T1 (credit ₹50) and T2 (credit ₹50) perform the credit operation on the same account A where the current balance of account A is ₹100:

Time	**T1**	**T2**	**Explanation**
t1	Read(A)		At t1 - T1 transaction reads the value of A, i.e., ₹100.
t2	A=A+50		At t2 - T1 transaction credits in A by ₹50
t3	Write(A)		At t3 - T1 Transaction writes the value of A (₹150) in the database.
t4		Read(A)	At t4 - T2 transaction read the value of A, i.e., ₹150
t5		A=A+50	At t5 - T2 transaction credits in A by ₹50
t6		Write(A)	At t6 - T2 Transaction writes the value of A (₹200) in the database.
t7	Power Failure + Rollback		At t7- T1 transaction fails due to power failure, then it is rolled back according to ACID properties. So, transaction T2 at t4 time reads a value that has not been committed in the database. The value read by the transaction T2 is known as a dirty read.

Table 6.2: *Dirty Read Problem Example*

- Unrepeatable Read Problem (Write-Read Conflict)

If in a transaction, two different values read for the same item stored in the database, the condition is known as "*Unrepeatable Read*

Problem". It is also known as Inconsistent Retrievals Problem.

Example: Consider Table 6.3, wherein two transactions, T1 and T2, are accessing account A, where the current balance of account A is ₹500:

Time	T1	T2	Explanation
t1	Read(A)		At t1 - T1 transaction reads the value of A, i.e., ₹500.
t2		Read(A)	At t2 - T2 transaction reads the value of A, i.e., ₹500.
t3	A = A + 500		At t3 - T1 Transaction credit in A by ₹500
t4	Write(A)		At t4 - T1 transaction writes the value of A, i.e., ₹1000
t5		Read(A)	At t1 - T2 transaction reads the value of A, i.e., ₹1000. So, here, transaction T2 reads two different values of account A; first it reads ₹500 and after the update is done by transaction T1, T2 reads it as ₹1000. It is an unrepeatable read and this condition is known as the Unrepeatable read problem.

Table 6.3: *Unrepeatable Read Problem Example*

Concurrency control systems of the database can solve these problems by using the locking techniques and the time at which a transaction modifies the data. This will be discussed in the following section on concurrency control mechanisms.

Concurrency Control

Concurrency control is the mechanism that ensures non-interference, isolation, and consistency of the database. It manages the concurrent execution of the transactions and produces accurate results without violating the ACID properties.

The following are the concurrency control protocols in DBMS which ensure the ACID properties and serializability of the concurrent execution of the database transactions:

- Lock-Based Protocols
- Two-Phase Locking Protocol
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-Based Protocols

This protocol is used in most commercial DBMSs. It guarantees serializability. In this type of protocol, all transactions must lock the data items before accessing them, which prevents multiple transactions from accessing the items concurrently.

A lock is a data variable that is associated with every data item available in the database. This lock indicates the type of operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager and the transactions can proceed only when the lock request is granted.

There are various types of lock protocols, which are as follows:

Binary Lock: A binary lock can have two values – locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X.

If the value of the lock on X is 1, it means that any transaction holding lock on item 'X' and item 'X' cannot be accessed by a database operation that requests the item.

If the value of the lock on X is 0, it means that it's free – the item can be accessed when requested.

Two operations, lock_item, and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item), and the transaction is allowed to access item X.

Shared Lock: It is also called a Read-only lock. In the shared lock, the data items can only be read by the transactions and are not allowed to update the data on the data item until the reading process is over by all the transactions. With the shared lock, the data item can be shared between any number of transactions with a read lock.

For example, consider a case of an online movie ticket booking, wherein two transactions are reading the available seats of any movie. The database will allow them to be read by placing a shared lock. However, if another transaction wants to update it by booking a few seats, the shared lock prevents it until the reading process is over.

Exclusive Lock: It is also called a read-write lock. In an exclusive lock, the data item can be both read as well as updated by the transaction. This lock is exclusive, which means, multiple transactions cannot update the same data simultaneously. Here, if any Transaction wishes to update the data item, it needs to hold an exclusive lock before accessing it. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to book the movie ticket, you can allow this transaction by placing an X lock on it. Therefore, when the second transaction wants to read or write, an exclusive lock prevents this operation.

Pre-claiming Lock Protocol: This protocol examines the transaction and makes a list of all the data items on which they need locks. Then, it makes a request to DBMS for all the locks required on those data items before initiating the execution of the transaction. After receiving grants for all the locks, this protocol allows the transaction to begin. When the transaction is complete, it releases all the locks.

Two-Phase Locking Protocol

It is also known as the 2PL protocol. 2PL is one of the concurrency control methods which ensure serializability in DBMS. However, it does not ensure that deadlocks do not happen. Basically, locks are held by the transaction which blocks the other transactions to access the same data simultaneously. 2PL helps eliminate the concurrency problem in DBMS.

This locking protocol divides the execution phase of a transaction into three different phases, which are as follows:

- In the first part, at the start of the transaction, it requests all the locks it needs during its execution.
- In the second part, the transaction gets the grants for all the locks. It enters into the third part when the transaction releases its first lock.
- In this third part, the transaction is not allowed for acquiring any new locks. It can only release the acquired locks.

Refer to *Figure 6.8* that illustrates the two-phase locking protocol:

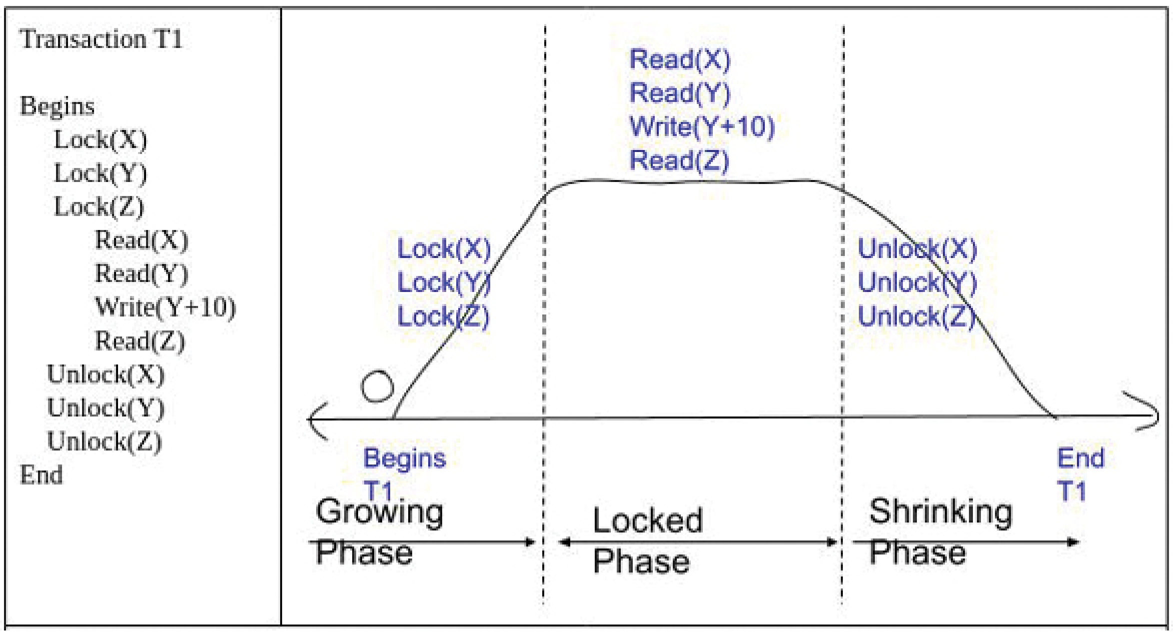


Figure 6.8: *Two-phase locking protocol*

There are two main phases of 2PL, which are as follows:

- **Growing phase**: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released. In the preceding *Figure 6.8*, data items X, Y, and Z are locked by transaction T1.

- **Shrinking phase**: In the shrinking phase, the existing locks held by the transaction may be released, but no new locks can be acquired. This can be seen in the preceding *Figure 6.8*, where transaction T1 is releasing the locks on data items X, Y, and Z.

Lock conversion is possible and the following phases can happen:

- Upgrading of lock (from Read lock to Write lock on any data item) is allowed in the growing phase.
- Downgrading of lock (from Write lock to Read lock of any data item) must be done in the shrinking phase.

There are three different variants of 2 phase locking protocols.

1. Basic
2. Strict
3. Rigorous

The basic 2PL has growing and shrinking phase. In growing and shrinking phase it won't release or acquire a lock on any data item.

In strict 2PL it will hold the write lock till the end and in the Rigorous 2 PL holds all locks till transaction commits or aborts.

Strict two-phase locking method

The strict-two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the transaction reaches the commit point and releases all the locks at that time. Hence, the Strict-2PL protocol does not have a shrinking phase of the lock release.

Timestamp ordering protocol

This protocol uses the concept of timestamp to order the transactions. Generally, the timestamp could be set as the system time or logical counter. The order of the transaction is used to arrange the execution of the concurrent transactions in the ascending order, which ensures that every conflicting read and write operation is executed in the timestamp order.

In this method, the older transaction is always given priority over the younger transaction. For example, assume there are three transactions, T1, T2, and T3. T1 enters the system at time 0010, T2 enters the system at time 0040, and T3 enters the system at time 0050. Here, T1 is the oldest transaction and it gets the highest priority, so it executes first. Similarly, T2 will get a chance to execute, and then T3, as it is the youngest transaction, gets executed last.

It also maintains the timestamp of the last 'read' and 'write' operation on the data items of the database, as follows:

- W-timestamp(X) – It means the largest timestamp of any transaction which executed the write operation on 'X' successfully.
- R-timestamp(X) – It means the largest timestamp of any transaction which executed the read operation on 'X' successfully.

These timestamps are updated whenever a new read or write operation is executed on data item 'X'.

Timestamp ordering protocol operates as follows:

- Suppose that transaction T1 wants to read 'X'; so it issues read(X).
    - If TS(T1) < W-timestamp(X), then T1 wants to read a value of X that was already overwritten. Hence, the read operation is rejected, and T1 has to be rolled back.
    - If TS(T1) ≥ W-timestamp(X), then the read operation is executed, and R-timestamp(X) is set to the maximum of R-timestamp(X) of TS(T1).
- Suppose that transaction T1 wants to write data item 'X', so it issues write(X).
    - If TS(T1) < R-timestamp(X), then the value of X that T1 is producing was needed previously, and the system assumed that value would never be produced. Hence, the system rejects the write operation and rolls back T1.

    - If TS(T1) < W-timestamp(X), then T1 is attempting to write an obsolete value of X. Hence, the system rejects this write operation and rolls back T1.
    - Otherwise, the system executes the write operation and sets W-time-stamp(X) to TS(T1).

Timestamp ordering protocol ensures serializability. It also ensures freedom from deadlock that means no transaction ever waits. But the schedule may not be recoverable and may not even be cascade-free.

Validation Based Protocol

Validation based protocol is also known as the Optimistic Concurrency Control Technique. In this technique, no checking is performed during the execution of the transaction. All updates happening during the execution of the transactions are not applied directly to the database but are applied to the local copies of the data items for the transactions. When a transaction reaches the end of the execution, checking takes place as to whether any of the transaction updates violate serializability. If there is no violation of serializability, then the transaction is committed and finally, the database is updated; otherwise, the transaction is restarted.

It is a three-phase protocol. The three phases for validation based protocol are as follows:

- **Read phase**: In this phase, transaction T reads the committed data items from the database and stores them in temporary local variables. Write operations can be performed on temporary variables without updating to the actual database.
- **Validation phase**: In this phase, checking is performed on temporary variable values to ensure that there is no violation of the serializability.
- **Write phase**: If there is no violation of the serializability, then updates on temporary local variable values are written to the database; otherwise, updates are discarded, and the transaction is rolled back.

To check the validity, the following time-stamps need to be assigned to transaction T1:

- **Start(T1)**: It is the time when T1 started its execution.

- **Validation(T1)**: It is the time when T1 just finished its read phase and began its validation phase.
- **Finish(T1)**: It is the time when T1 completes all its writing operations in the database during the write-phase.

Two more terms that we need to know are as follows:

- **Write_set:** This set contains all the write operations that T1 performs.
- **Read_set:** This set contains all the read operations that T1 performs.

Timestamp for the transaction can be determined for serialization using the timestamp of the validation phase, as it is the actual phase that determines if the transaction will commit or rollback. The serializability can be determined during the validation process. It can't be decided in advance.

The validation phase for T1 checks that for all Transactions T2, one of the following conditions must hold for passing the validation phase:

- Finish(T2)<Starts(T1): Here, T2 finishes its write-phase before T1 starts its read-phase. So, they do not violate serializability.
- T1 begins its write phase after T2 completes its write phase, and the read_set of T1 should be disjoint with the write_set of T2. Then, the serializability is indeed maintained.
- T2 completes its read phase before T1 completes its read phase and both read_set and write_set of T1 are disjoint with the write_set of T2. The serializability order is indeed maintained.

Deadlock

In DBMS, deadlock is an undesirable state which generally occurs in a shared resources environment, where a transaction waits indefinitely for a resource that is held by another transaction. Thus, all the transactions wait for each other to release the resources, and none are able to finish their task. This condition is known as '*Deadlock*', as shown in *Figure 6.9*:

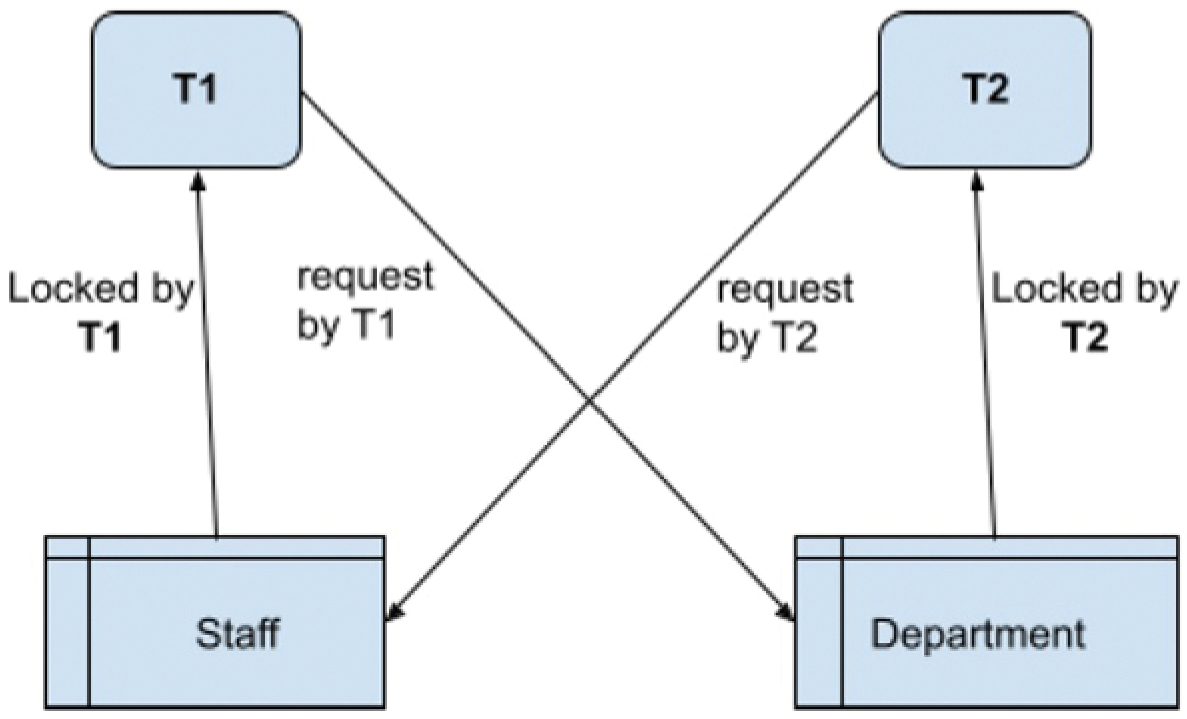


Figure 6.9: *An Example of Deadlock*

Let us understand the Deadlock concept with the following example:

Assume two transactions T1 and T2. Suppose, Transaction T1 holds a lock on the Staff table and needs to add some rows in the Department table. At the same time, Transaction T2 holds a lock on the Department table but wants to access a few rows of the Staff table held by Transaction T1, as shown in *Figure 6.9*. In this situation, both the transactions wait for each other to release the lock on the tables and no one is able to finish their task. As a result, they are waiting indefinitely for a resource and remain idle forever unless DBMS detects it as a deadlock and takes appropriate action such as aborting one of the transactions.

Deadlock handling methods

DBMS handles deadlock by one of the following methods:

- Deadlock avoidance
- Deadlock prevention
- Deadlock detection

Deadlock Avoidance

A simple technique to handle the deadlock is to avoid it. The deadlock occurs very rarely. If we train the system for handling the deadlock, the performance and speed of the system will degrade. That is the reason, most of the systems ignore the deadlock situation as they are more concerned with the performance and speed of the system.

Deadlock Prevention

For a large database, a deadlock prevention method is more suitable. Assign resources in such a way that it never causes deadlock to happen, thus the deadlock can be prevented.

The DBMS analyzer should predict whether the transactions can create a deadlock in the future or not. If so, that transaction is aborted.

We should always prevent any or all of the following conditions from holding true; if we fail, a deadlock may occur:

- **Removal of mutual exclusion**: All resources must be sharable which means, at a time, more than one process can get access to the resources. Practically, this approach is impossible.
- **Removal of hold and wait condition**: This approach is quite possible; before starting the process, it should acquire all the resources that are needed in advance. This is what happens in the conservative two-phase locking protocol.
- **Preemption of resources**: If we forcibly preempt or stop a transaction, then the resources held by one transaction will be released and can be used by another transaction and deadlock can be removed (Wound-Wait scheme).
- **Avoid circular wait conditions**: Resources can be maintained in a hierarchy and the processes can hold the resources in an increasing order of precedence to avoid a circular wait.

Wait-Die and Wound-Wait are the two schemes that are used in the timestamp ordering protocol in order to predetermine a deadlock situation.

Let us understand each one in detail.

Wait-die scheme

In this scheme, if a transaction requests for a resource, which is already locked by another transaction, then one of the two following conditions may arise:

- TS(T1) < TS(T2): Here, T1 and T2 are the two transactions, and the timestamp of any transaction T can be denoted as TS (T). Here, if T1 is the older transaction and T2 is the younger transaction and holds some resources, then the older transaction T1 is allowed to wait for the requested resource until it is available.
- TS(T1) > TS(T2): Here, T1 is the younger transaction and T2 is the older transaction that holds some resource. Since T1 is younger and waiting for the resource held by older transaction T2, T1 is killed and restarted later with the same timestamp with random delay.

This scheme allows the older transaction to wait and kills the younger one.

Wound-wait scheme

In this scheme, if a transaction requests for a resource, which is already locked by another transaction, then one of the two following conditions may arise:

- TS(T1) < TS(T2): Here, if T1 is an older transaction and requesting for a resource held by a younger transaction T2, then T1 preempts the resource held by T2 by killing it. When T2 gets killed, it will be restarted later with the same timestamp with random delay.
- TS(T1) > TS(T2): Here T1 is the younger transaction and requested a resource which is held by the older transaction T2. So T1 is asked to wait till T2 releases it.

In this scheme, it allows the younger transaction to wait, until the older transaction requests for a resource held by a younger one. If an older transaction needs a resource, it preempts the resource held by the younger transaction by aborting it. In certain cases, it may happen that both these schemes may result in needless aborting and restarting of the younger transactions.

Deadlock Detection

If deadlock avoidance and prevention are not taken care of properly at an earlier stage, then a deadlock may occur. So, we need to detect the deadlock in advance and try to recover it.

Wait-for-graph is the simplest method for detecting the deadlock situation. In this method, a graph is drawn based on the transaction and their lock on the resource. If the graph created has a closed-loop or a cycle, then there is a deadlock, as shown in *Figure 6.10*:

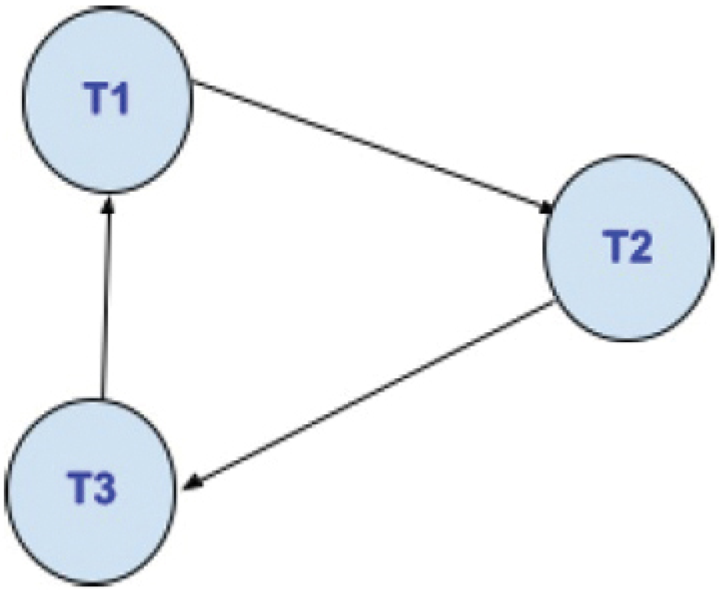


Figure 6.10: *An Example of Wait-for-graph*

In the preceding graph, the vertices represent the transactions and the directed edge from a transaction T1 to T2 indicates that T1 is waiting for a resource that is held by T2. Similarly, T2 is waiting for a resource that is held by T3 and T3 is waiting for a resource that is held by T1. As a result, this graph contains a cycle, which detects that there is a Deadlock.

Transaction Control Commands

To perform and manage the transactions in the database, the transaction control language commands are used. The COMMIT, ROLLBACK, and SAVEPOINT statements are also called the transaction control language (TCL) statements. These commands manage the changes made by the DML statements, as follows:

- **COMMIT:**

  The commit command is used to permanently save the changes done by the user into the database. In the logical transactions, the final command issued is the commit command.

- **ROLLBACK:**

  The rollback command is just opposite to the Commit command. It restores the database to the last committed state. Rollback performs the undo operation which was carried out during the logical transaction processing. It undoes the operations till the last commit command.

- **SAVEPOINT:**

  Transaction Control Language (TCL) commands manage the logical transactions in the database. The DML statements make the changes to the data. A set of DML statements/commands form a logical transaction, wherein the SAVEPOINT command is used to save the transaction temporarily. In the case of rollback, changes done till SAVEPOINT will be unchanged, that is, the transaction can be rolled back to that point whenever required.

Recovery System

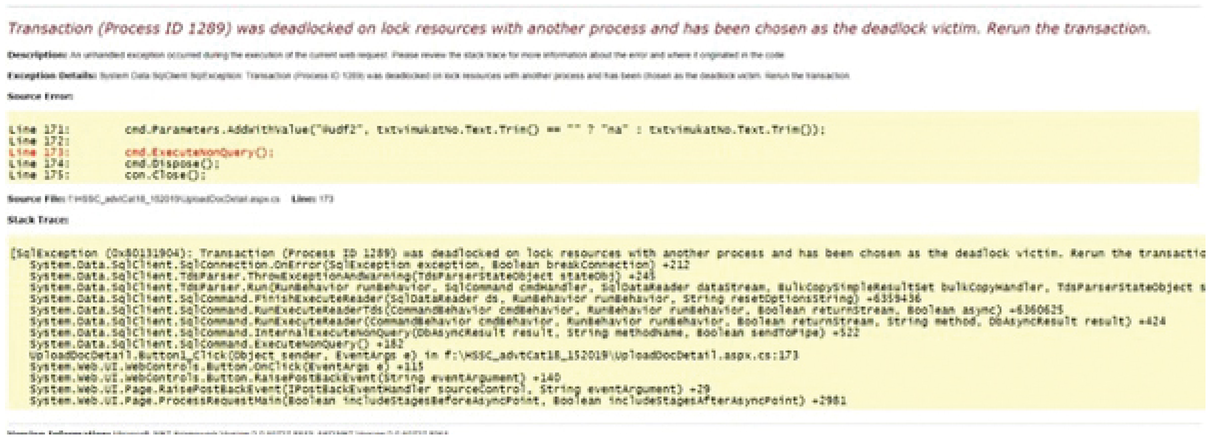
Transactions (or units of work) against a database can be interrupted unexpectedly because of the failure for a variety of reasons. Understanding the concepts of database recovery requires a clear comprehension of the type of failure that the database has to cope with and the notion of consistency that is assumed as a criterion for describing the state to be reestablished. If a failure occurs before all of the changes that are part of the unit of work are completed, committed, and written to disk, the database is left in an inconsistent and unusable state. Crash recovery is the process by which the database is maintained to the consistent state that existed before the failure. In particular, we are establishing a systematic framework for establishing and evaluating the basic concepts for the fault-tolerant database operation.

The concept of a transaction (or units of work) includes all database interactions between $BEGIN_TRANSACTION and $COMMIT_TRANSACTION. If we consider an example of a transaction, then it requires that all of its actions be executed indivisibly – either all actions are properly reflected in the database or nothing has happened. To achieve this kind of indivisibility, a transaction must have the ACID properties i.e. atomicity, consistency, isolation, and durability, as described as follows:

- **Atomicity**: It must be of the all-or-nothing and the end-user must know which state he or she is in.
- **Consistency**: A transaction reaching its normal end of a transaction, thereby committing its results, preserves the consistency of the database.
- **Isolation**: Operations within a transaction must be hidden from the other transactions running concurrently.
- **Durability**: Once a transaction has been completed and has committed its results to the database, the system must guarantee that these results persist.

A transaction can end in the following three ways:

- It can reach its commit point.
- If it violates the consistency, thus preventing a normal termination, it gets aborted.
- A transaction may run into a problem that can only be detected by the system, such as time-out or deadlock, in which case, its effects are aborted by the DBMS, as shown in *Figure 6.11*:



Transaction (Process ID 1289) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Transaction (Process ID 1289) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Source Error:

```
Line 171:        cmd.Parameters.AddWithValue("@udf2", txtvimukatNo.Text.Trim() == "" ? "na" : txtvimukatNo.Text.Trim());
Line 172:
Line 173:        cmd.ExecuteNonQuery();
Line 174:        cmd.Dispose();
Line 175:        con.Close();
```

Source File: f:\HSSC_advtCat18_152019\UploadDocDetail.aspx.cs   **Line:** 173

Stack Trace:

```
[SqlException (0x80131904): Transaction (Process ID 1289) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transactio
   System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean breakConnection) +212
   System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj) +245
   System.Data.SqlClient.TdsParser.Run(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader dataStream, BulkCopySimpleResultSet bulkCopyHandler, TdsParserStateObject s
   System.Data.SqlClient.SqlCommand.FinishExecuteReader(SqlDataReader ds, RunBehavior runBehavior, String resetOptionsString) +6359436
   System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(CommandBehavior cmdBehavior, RunBehavior runBehavior, Boolean returnStream, Boolean async) +6360625
   System.Data.SqlClient.SqlCommand.RunExecuteReader(CommandBehavior cmdBehavior, RunBehavior runBehavior, Boolean returnStream, String method, DbAsyncResult result) +424
   System.Data.SqlClient.SqlCommand.InternalExecuteNonQuery(DbAsyncResult result, String methodName, Boolean sendToPipe) +522
   System.Data.SqlClient.SqlCommand.ExecuteNonQuery() +182
   UploadDocDetail.Button1_Click(Object sender, EventArgs e) in f:\HSSC_advtCat18_152019\UploadDocDetail.aspx.cs:173
   System.Web.UI.WebControls.Button.OnClick(EventArgs e) +115
   System.Web.UI.WebControls.Button.RaisePostBackEvent(String eventArgument) +140
   System.Web.UI.Page.RaisePostBackEvent(IPostBackEventHandler sourceControl, String eventArgument) +29
   System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAfterAsyncPoint) +2981
```

Version Information: [illegible]

Figure 6.11: *Transaction aborted by the DBMS because of deadlock.*

In addition to the preceding reasons or events occurring during normal execution, a transaction can also be affected by a system crash.

In order to design and implement a recovery system, one must consider the following points:

- Know precisely which types of failures are to be considered
- The frequency of occurrence and their dependencies
- The expected time required for recovery

Making assumptions about the reliability of the underlying hardware, storage media, and various other failures will be very difficult as some of the failures are extremely rare and some of them are catastrophic. Broadly, we shall consider the transaction failure, system failure, and media failure.

Let us consider an example, as shown in Figure 6.12, for discussing the transaction-oriented recovery:

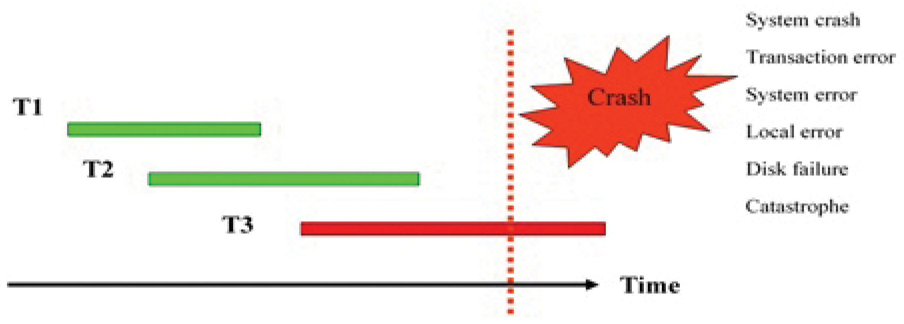


Figure: 6.12: *Scenario for discussing transaction-oriented recovery.*

In case of consistency that we use for defining, the targets of recovery is tied to the transaction paradigm, which we have encapsulated in the "*ACID principle*". According to this definition, a database is consistent if, and only if, it contains the results of successful transactions, that is, a transaction, by definition, creates a consistent update of the database. What does that mean for the recovery component?

Let us consider only a system failure (a crash), as shown in Figure 6.12. Transactions T1 and T2 have committed before the crash and, therefore, will survive. Recovery after a system failure must ensure that the effects of all the successful transactions are actually reflected in the database. But what is to be done with T3? As the transactions have been defined to be atomic, they either succeed or disappear as though they had never been entered. So, if required, the transactions T1 and T2 can be aborted, as the durability property of ACID says the applied changes by the transaction must not be lost.

We now can distinguish the two main recovery actions, i.e., the process by which the database is restored to the most recent consistent state just before the time of failure considering the different situations. If there is extensive damage, then restore the database entirely from a backed-up copy; otherwise, reverse any changes that caused the inconsistency either by undoing or redoing.

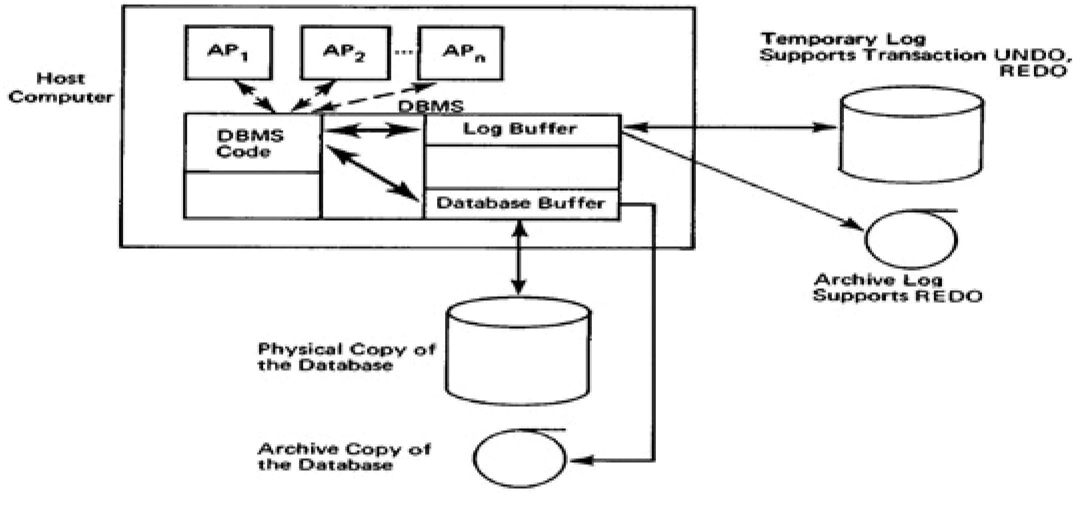

Host Computer
AP₁
AP₂
APₙ
DBMS
DBMS Code
Log Buffer
Database Buffer
Temporary Log Supports Transaction UNDO, REDO
Archive Log Supports REDO
Physical Copy of the Database
Archive Copy of the Database

Figure 6.13: *Storage hierarchy of a DBMS during the normal mode of operation (ref-THEO HAERDER and ANDREAS REUTER)*

- **Transaction UNDO**: If a transaction aborts itself or must be aborted by the system during normal execution, this will be called "*transaction UNDO*." By definition, UNDO removes all effects of this transaction from the database and does not influence any other transaction. In case of a system failure, the effects of all incomplete transactions have to be rolled back during the recovery process.
- **Transaction REDO**: When recovering from a system failure, since the execution has been terminated in an uncontrolled manner, the results of the complete transactions may not yet be reflected in the database. Hence, they must be repeated, if necessary, by the recovery system. Considering the transaction as the unit of recovery and their existence typically for a short duration, we can supplement the most recent copy with the effects of all the transactions that have been committed since the copy was created.

The recovery process works closely with the operating system for buffering or caching of pages in **main memory** (**MM**) and collecting such data is strongly influenced by the various properties of the different storage media used by the DBMS. Let us consider that we shall use a storage hierarchy, as shown in *Figure 6.13* (THEO HAERDER and ANDREAS REUTER).

As we know, the main memory is volatile and we will lose the contents of the database buffer and the output buffers to the log files, which are lost whenever there is a system failure and DBMS terminates abnormally. So, to recover from system failure, we can use the permeant storage (secondary storage), or during a crash, it might be recovered from the archive copy as a provision against loss of the online copy. The recovery system looks into the transaction log files in order to recover from the failure to the most recent database state before the failure and can perform the selective UNDO operations. These log files are stored on the disk, as shown in *Figure 6.14*:

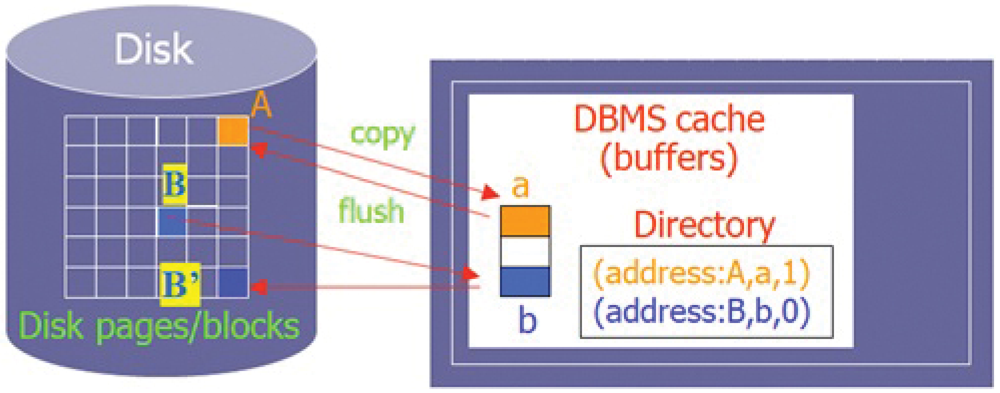


Figure 6.14: *Disk pages and DBMS cache.*

Figure 6.14 shows the database buffers which maintain the cache of current pages in the main memory. These pages need to be replaced from disk to the main memory or vice versa based on the requirement of the data to perform the transactions. To perform these actions, either it will get copied on the disk at the same place or it might maintain a separate copy of the data blocks. Here, the assumption is that each page of a segment is related to exactly one block of the corresponding file. Each output of a modified page causes an update in place. If the pages are flushed to different blocks, then the old contents of the page remain unchanged and it is possible to maintain the different versions of the pages. The modified pages may be written to the disk by FIFO (First In First Out) or LRU (Least Recently Used) policies.

- **Buffer management and UNDO recovery actions**: To keep track of the modified pages of the incomplete transaction and to limit the UNDO operations to the main storage, the Dirty bit (0 or 1) can be associated with each block and it indicates if the buffer has been modified, and the pin/unpin bit (0 or 1) associated with each block indicates if the buffer can (or cannot) be written back to disk. To avoid writing of large database buffers because of this strategy and for better handling of the modified pages with UNDO recovery, the STEAL No STEAL method is used. In STEAL case, the updated or dirty pages may be written any time and in the case of NO STEAL, the dirty pages are kept in buffer at least until the transaction reaches the end, i.e., EOT. This makes it clear that in the case of NO STEAL, no logging is required for UNDO purposes.

- **Buffer management and REDO recovery actions**: The durability property of the transactions ensures that the results of the committed transaction must persist during the subsequent failure. If the results of the committed transactions are not stored on the disk, then the database would definitely be lost in case of a system crash; so there must be enough information in the log file to reconstruct these results during restart. i.e., REDO.

Can we avoid this REDO recovery?

If all modified pages are stored on the disk during the EOT phase, that is, their writing on the disk is enforced, and we are sure that all the results are safely stored on the disk, then there is no need for REDO. If the transaction is not successfully completed, then the transaction is not successful and must be rolled back (UNDO recovery actions). This shows that we have another criterion for buffer management handling, which is related to the necessity of REDO recovery during restart. The standard recovery terminology of database management system includes the terms force/no-force, to indicate when a page from the database can be written to disk from the cache:

- **FORCE**: All modified pages are written to disk during EOT processing.
- **No-FORCE**: No disk write operation is triggered during EOT processing.

So from this, it is clear that FORCE does not require the gathering of log data but in the case of NO-FORCE, the log data must be maintained to perform the REDO operations to make the database consistent. This indicates that the type of write operation on the disk decides the need for log data for the purpose of REDO operation to maintain the consistency and remove the unwanted changes of the uncommitted transactions. Now, let us describe the log data to understand what data is required to maintain the consistency of the database using *Figure 6.15*.

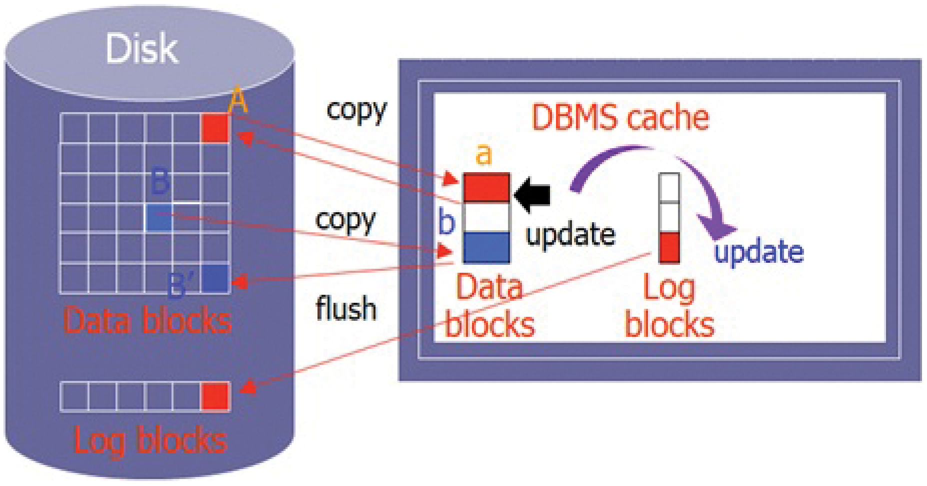


Figure 6.15: *Disk Blocks, Log Blocks, and DBMS Cache*

The log data can be created using physical logging or logical logging. In the case of physical logging, the bit pattern is written to the log. If the operators and their arguments are recorded on a higher level, this is called logical logging. The physical logging works at the page level and whenever there is some modification, the page is written to the log, as shown in the *Figure 6.15*. If the UNDO operation is to be performed, then we require the before image and if the REDO operations are needed to be performed for recovery, then the after image, that is, the modified page is required. In fact, the changes required for UNDO must be written to disk before these changes are applied to the database to recover from failure. This concept is referred to as the **Write-Ahead Logging** (**WAL**) principle. Write-ahead Logging is used with a recovery technique in-place updating and writing BFIM (old item) in the log entry on DISK and changing old (BFIM) by new (AFIM) on Database. For a write operation, there would be an UNDO-type entry (holds BEIM), and a REDO-type entry (holds AFIM). REDO information must be written to the temporary and archive log file before EOT is acknowledged to the transaction program. Once this is done, the system must be able to ensure the transaction's durability.

For a log-based recovery system, the main question is, to what extent should the log be processed for UNDO recovery. Similarly, for REDO, if the DBMS does not use a FORCE discipline, the question is, which part of the log has to be processed for REDO recovery.

The UNDO case is straightforward, as we need to scan the log file back to the **Beginning of Transaction** (**BOT**) entry of the oldest incomplete transaction to be sure that no invalid data are left in the system. But the REDO process is not as straightforward. If we can assume that in the case of a FORCE discipline, modified pages will be written eventually because of buffer replacement, one might expect that only the contents of the most recently changed pages have to be redone if the change was caused by a complete transaction. But, in the case of large database applications, most of the modified pages will have been changed "recently" but there are a few frequently used pages that are modified again and again, and since they are referenced so frequently, they have not been written from the buffer (the page replacement policies). After a while, such pages will contain the updates of many complete transactions, and REDO recovery will, therefore, have to go back very far on the temporary log. This makes the restart expensive. In general, the amount of log data to be processed for REDO will increase with the interval of time between the two subsequent crashes.

In other words, the higher the availability of the system, the more costly the recovery will become. This brings the concept of checkpoints. Generating a checkpoint means collecting information in a safe place, which has the effect of defining and limiting the amount of REDO recovery required after a crash. The checkpoint has three steps – BEGIN_CHECKPOINT, write all checkpoint data to the log file and/or the database, and END_CHECKPOINT. The checkpoint technique is for optimizing crash recovery. So, it must be generated at well-defined points in time. The checkpoints are written into the log periodically when the system writes out to the database on disk for all modified DBMS-buffers.

Taking a checkpoint consists of the following:

- Suspending active transactions temporarily
- Force-write all modified MM-buffers to the database on disk
- Write [Checkpoint] entry to the log
- Force-write the log to disk
- Resume executing transactions

Refer to *Figure 6.16* that illustrates an example of checkpoint:

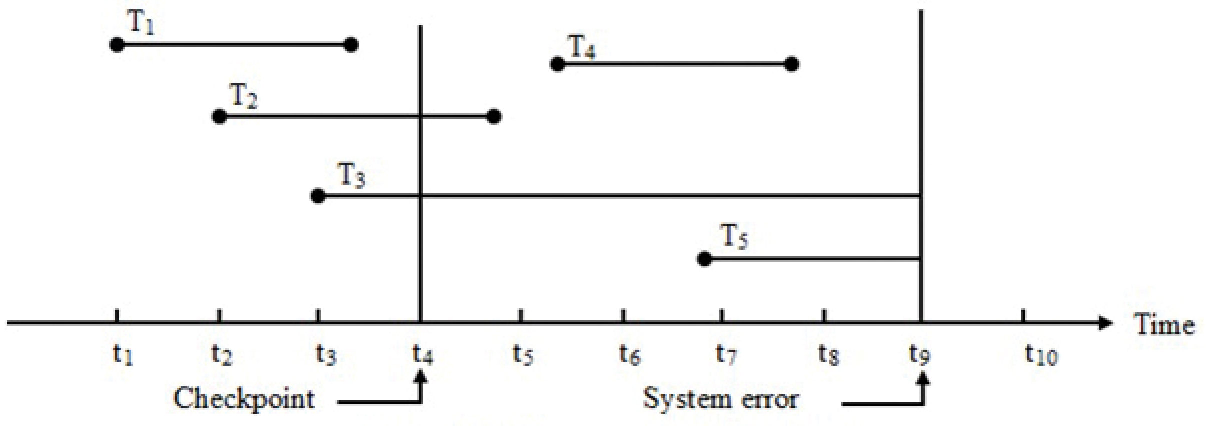


Figure 6.16: An example of Checkpoint

Figure: 6.16: *An example of checkpoint*

Figure 6.16 illustrates the concept of checkpoints. When the checkpoint was taken at time t4, transaction T1 were committed, whereas transactions T2 and T3 were not. Before the system error occurs at time t9, transaction T2 and T4 were committed but transaction T3 and T5 were not. According to the RDU_M method, there is no need to redo the write_item operations of transaction T1—or any transactions committed before the last checkpoint time t4. However, the write_item operations of T2 and T4 must be redone because both the transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions T3 and T5 are ignored. They are effectively canceled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol.

Conclusion

Transaction is a logical unit of work that represents a real-world entity, whereas concurrency control is the management of concurrent transaction execution.

A transaction has ACID properties – Atomicity, Consistency, Isolation, and Durability.

A transaction can be in any one of the following states:

- Active state

- Partially committed state
- Committed state
- Failed state
- Aborted state
- Terminated state

In concurrent executions, multiple transactions run concurrently for improving throughput, resource utilization, and reducing waiting time. But it may lead to the following problems:

- Lost update problem
- Dirty read problem
- Unrepeatable read problem

When a transaction waits indefinitely for a resource that is held by another transaction, it's in an undesirable state which is known as deadlock.

Deadlock can be handled using the following methods:

- Deadlock avoidance
- Deadlock prevention
- Deadlock detection

The transaction control language commands are used to perform and manage the transactions in the database; they are COMMIT, ROLLBACK, and SAVEPOINT.

Concurrency control is the mechanism that ensures non-interference, isolation, and consistency of the database.

Concurrency control protocols in DBMS ensure the ACID properties and serializability of the concurrent execution of the database transactions. They are as follows:

- Lock-based protocols
- Two-phase locking protocol

- Timestamp-based protocols
- Validation-based protocols

Lock-based protocols are used to manage the order between the conflicting pairs among the transactions at the time of execution.

Timestamp-based protocols use the concept of Timestamp. Every transaction has a timestamp associated with it, and the ordering is determined by the value of the Timestamp of the transaction. Older transactions get the highest priority for execution.

Validation based protocol is also known as the Optimistic Concurrency Control Technique. It is used for avoiding deadlock in transactions.

Recovery is the process by which the database is restored to the most recent consistent state just before the time of failure.

Schedule refers to the order of operations from multiple transactions and is only interested in read (r), write (w), commit (c), and abort (a) operations.

A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions. The two schedules are equivalent if they are view equivalent or conflict equivalent.

Two schedules, S0 and S1, are conflict equivalent if S0 can be transformed into S1 by a series of swaps of non-conflicting operations.

Questions

1. What is a transaction? What are its properties? Why are the transactions important units of operation in a DBMS?
2. Which are the basic operations in Transaction?
3. Draw a state diagram of a Transaction and discuss each state that a transaction goes through during its execution.
4. List the ACID properties.
5. What do you mean by atomicity? Why is it important? Explain with an example.
6. What do you mean by consistency? Why is it important? Explain with an example.

7. What are the problems of concurrency in transactions?
8. Define the Dirty read problem.
9. Explain the concurrency control techniques.
10. What are the different types of locks?
11. Write an algorithm for a shared lock.
12. Explain two-phase locking protocols. Write its advantages and disadvantages.
13. What is the two-phase locking protocol?
14. Define Strict 2PL.
15. How does upgrading and downgrading of the locks help in the concurrency of transactions?
16. What is a deadlock? How can a deadlock be avoided?
17. List the different deadlock prevention schemes.
18. Define the wait-die protocols.
19. What is timestamp ordering? What are the variants of timestamp ordering?
20. What are the drawbacks of a timestamp?
21. Explain the basis of Validation based concurrency control protocols.
22. List the advantages, problems, and application of the optimistic methods of concurrency control.
23. Which of the concurrency control protocols ensure both conflict serializability and release from deadlock?
24. Consider the following log sequence of two transactions on a bank account, with an initial balance of 12000, that transfers 2000 to a mortgage payment and then applies a 5% interest:

    1. T1 start
    2. T1 B old=12000 new=10000
    3. T1 M old=0 new=2000
    4. T1 commit
    5. T2 start
    6. T2 B old=10000 new=10500

7. T2 commit

Suppose the database system crashes just before log record 7 is written. When the system is restarted, which one statement is true of the recovery procedure?

A. We must redo log record 6 to set B to 10500
B. e must undo log record 6 to set B to 10000 and then redo log records 2 and 3
C. We need not redo log records 2 and 3 because transaction T1 has committed
D. We can apply redo and undo operations in arbitrary order because they are idempotent

25. Check whether the following schedule S is conflict serializable or not:

    S : R1(A) , R2(A) , R1(B) , R2(B) , R3(B) , W1(A) , W2(B)

26. Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

    A: r1(X);r3(X);w1(X);r2(X);w3(X);

    B: r1(X);r3(X);w3(X);w1(X); r2(X);

    C: r3(X);r2(X);w3(X);r1(X);w1(X);

    D: r3(X);r2(X);r1(X);w3(X);w1(X);

27. Check whether the following schedule S is view serializable or not. If yes, then give the serial schedule.

    S : R1(A) , W2(A) , R3(A) , W1(A) , W3(A)

28. Explain how the wait for graphs can be used for deadlock detection.

29. Our database is running into problems, as many transactions are waiting for locks held by the other transactions. The following is what each transaction is waiting for:

    - T1 is waiting on T4
    - T2 is waiting on T7
    - T3 is waiting on T2
    - T4 is waiting on T1
    - T5 is waiting on T8

- T6 is waiting on T2
- T7 is waiting on T6
- T8 is not waiting

Draw the Wait-For graph for transactions T1−T8.

30. What is a transaction log and how is it used in recovery?
31. Are there any disadvantages of time-stamping methods for concurrency control? Explain with suitable examples.
32. Explain the use of binary and shared/exclusive locks in a DBMS.
33. How does DBMS resolve the conflicts?
34. What is checkpoint and how is it used for recovery?

Index