Dbms (4331603) - Problem Solver

Antigravity AI

June 4, 2026

Dbms

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Database System Concepts and Architecture

1.1 Data Information and Processing

Methodology:

Data Processing Cycle The transformation of raw facts (Data) into meaningful insights (Information) follows a systematic processing cycle:

1. **Input:** Collect raw facts and figures (Data).
2. **Processing:** Apply computational logic mathematical operations filtering or aggregation to the raw data.
3. **Output:** Generate structured organized context-rich results (Information) for decision making.

Worked Example:

Computing Student Grades **Problem:** Given raw data consisting of a student's marks in three subjects (45 55 60) out of 100 process this data to determine the total percentage and the final grade (Information).
Solution:

1. **Input Data:** Marks = {45, 55, 60}.
2. **Processing Computation:**
   - Total Marks = $45 + 55 + 60 = 160$
   - Percentage = $(160/300) \times 100 = 53.33\%$
   - Grade Logic: If Percentage $> 50\%$ Grade = Pass else Fail.
3. **Output Information:** Student Percentage is 53.33% and Grade is Pass.

1.2 Database Characteristics and Basic Concepts

Methodology:

Defining Schema and Instances A Database Management System (DBMS) structures data using schemas and stores dynamic states using instances.

1. **Schema (Intension):** Define the logical structure design and constraints of the database. It is static and rarely changes.
2. **Instance (Extension):** Insert data into the database at a specific moment in time. It is dynamic and changes with every insert update or delete operation.
3. **Metadata and Data Dictionary:** Store the definitions of schemas data types constraints and relationships (i.e. data about data).

Worked Example:

Identifying Intension and Extension **Problem:** Consider a table designed to store Employee details. Define its Schema (Intension) Metadata and Instance (Extension).
Solution:

1. **Schema (Intension):** `Employee(EmpID Integer, Name String, Salary Float)`
2. **Metadata:** The Data Dictionary stores rules such as "`EmpID` is a Primary Key of type Integer and cannot be NULL".
3. **Instance (Extension):** At $t = 1$ the table contains the following data records:

EmpID	Name	Salary
101	Alice	50000.00
102	Bob	60000.00

1.3 Architecture of DBMS

Methodology:

Selecting DBMS Architecture Evaluate the deployment environment to choose between Two Tier and Three Tier Architectures:

1. **Two Tier Architecture:** Client applications communicate directly with the database server. Use for local networks with limited users where application logic runs on the client machine.
2. **Three Tier Architecture:** Introduces an intermediate Application Server between the Client and Database. Use for web applications requiring high security scalability and centralized business logic.

Worked Example:

Architecture Selection for Web Application **Problem:** An e-commerce platform expects 10000 concurrent users accessing a centralized product database over the Internet. Select and justify the DBMS architecture.
Solution:

1. **Analysis:** High concurrent users and Internet access demand security scalability and business logic separation. A Two Tier approach would expose the database directly to the Internet.
2. **Selection:** Three Tier Architecture.
3. **Structure Application:**
   - *Presentation Tier:* Web Browser or Mobile App (Client).
   - *Application Tier:* Web Server running business logic (e.g. Node.js Java).

- *Database Tier:* DBMS server handling data retrieval and transactions.

1.4 File Oriented System vs DBMS

Methodology:

Resolving Data Anomalies To migrate from a File Oriented System to a Relational DBMS apply the normalization process to eliminate anomalies:

1. **Data Redundancy:** Identify and remove duplicate data stored across multiple files.
2. **Inconsistency:** Ensure that updating data in one place reflects everywhere automatically by using relationships instead of duplicated fields.

Worked Example:

Converting Flat File to DBMS **Problem:** A file system uses a single flat file for Students and their Departments leading to redundant Department Names. Convert this to a DBMS approach.
Solution:

1. **Flat File Record Structure:** `[StudentID, StudentName, DeptID, DeptName]`
   Example Records: `[1, "John", 10, "Computer"], [2, "Mary", 10, "Computer"]`
2. **DBMS Conversion Strategy:** Split into two related tables linked by a Primary/Foreign key relationship.
3. **Table 1 (Department):** `Dept(DeptID, DeptName)`
4. **Table 2 (Student):** `Student(StudentID, StudentName, DeptID)` (where DeptID is a Foreign Key).
5. **Result:** The redundancy of the word "Computer" is eliminated. Updating the department name now requires only one change in the `Dept` table.

1.5 Database Users

Methodology:

Role Assignment Protocol Assign database access and privileges based on user roles and required tasks:

1. **Database Administrator (DBA):** Grants permissions performs backups creates schemas and optimizes performance.
2. **Application Programmer:** Writes backend application code to interact with the database using APIs and queries.
3. **End User:** Uses the application interface to query and update data without knowing any database internals.

Worked Example:

Mapping Tasks to Database Users **Problem:** Given the following tasks assign the appropriate database user role: (A) Create new database tables. (B) Write Java code to fetch data. (C) Click a button to view a monthly report.
Solution:

1. **Task A (Create new tables):** Assigned to **DBA** (Requires schema modification and administrative

privileges).

2. **Task B (Write Java code to fetch data):** Assigned to **Application Programmer** (Requires coding and DML integration skills).

3. **Task C (Click button to view report):** Assigned to **End User** (Interacts strictly via GUI and requires no technical knowledge).

1.6 Data Independence

Methodology:

Achieving Data Independence Utilize the Three Schema Architecture to ensure changes at lower levels do not affect higher levels:

1. **Logical Data Independence:** Modify the conceptual schema (e.g. adding a new column or table) without changing external views or application programs.

2. **Physical Data Independence:** Modify the internal schema (e.g. changing storage media indexing or file organization) without changing the conceptual logical schema.

Worked Example:

Demonstrating Logical Data Independence **Problem:** An application uses an External View that displays `StudentName` and `Grade`. The DBA needs to add a `DateOfBirth` column to the underlying `Student` table. Demonstrate how Logical Data Independence is maintained.
Solution:

1. **Original Schema:** `Student(ID, Name, Grade)`

2. **External View Definition:** `SELECT Name, Grade FROM Student;`

3. **Schema Modification:** The DBA executes an `ALTER TABLE` command to add the `DateOfBirth` attribute.

4. **New Schema:** `Student(ID, Name, Grade, DateOfBirth)`

5. **Validation:** The External View query still successfully retrieves `Name` and `Grade`. The application program does not break because it is isolated from structural additions at the conceptual level.

1.7 DBMS Languages DDL DML DCL DQL

Methodology:

Executing Database Commands Use standard SQL language subsets to define manipulate and secure database structures:

1. **DDL (Data Definition Language):** Define and modify schema structures (`CREATE, ALTER, DROP, TRUNCATE`).

2. **DML (Data Manipulation Language):** Modify instance data (`INSERT, UPDATE, DELETE`).

3. **DQL (Data Query Language):** Retrieve data from tables (`SELECT`).

4. **DCL (Data Control Language):** Manage security and access rights (`GRANT, REVOKE`).

Worked Example:

Creating and Manipulating a Table **Problem:** Write the computational SQL commands to create a `Course` table insert one record and retrieve it.
Solution:

1. **Step 1: DDL Command (Define Schema)**

```
CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(50)
);
```

2. **Step 2: DML Command (Define Extension Instance)**

```
INSERT INTO Course (CourseID, CourseName)
VALUES (101, 'Database Systems');
```

3. **Step 3: DQL Command (Data Retrieval)**

```
SELECT * FROM Course;
```

1.8 Data Models

Methodology:

Applying the Relational Data Model Represent entities and their relationships using mathematical relations (tables):

1. **Relation:** A two-dimensional table consisting of rows and columns.
2. **Tuple:** A single row representing one specific entity instance.
3. **Attribute:** A column representing a specific property of the entity.
4. **Domain:** The set of valid allowable values for a given attribute.

Worked Example:

Designing a Relational Schema **Problem:** Model a "Library" system containing Books and Publishers using the Relational Data Model. A Publisher can publish multiple Books.
Solution:

1. **Identify Entities:** Book and Publisher.
2. **Define Publisher Relation:**
   - Schema: `Publisher(PubID, PubName, City)`
3. **Define Book Relation:**
   - Schema: `Book(BookID, Title, Price, PubID)`
   - Note: `PubID` in the `Book` relation acts as a Foreign Key to link to the primary key in the `Publisher` relation establishing a one-to-many relationship.

CHAPTER 2

Entity Relationship Diagram

2.1 Basic concepts (Entity, Attribute, Relationship)

Methodology:

Identifying Basic ER Components To extract Entity-Relationship (ER) model components from a problem statement:

1. **Identify Entities:** Look for nouns representing independent objects or concepts (e.g. Student, Course).
2. **Identify Attributes:** Look for properties or characteristics associated with the entities (e.g. Name, Roll Number).
3. **Identify Relationships:** Look for verbs connecting entities representing interactions (e.g. "enrolls in", "teaches").

Worked Example:

University Database Requirements **Problem:** A university maintains data about students and courses. Each student has a roll number, name, and age. Each course has a course ID and course name. Students enroll in courses. Identify the entities, attributes, and relationships.
Solution:

1. **Entities:**
   - `Student`
   - `Course`
2. **Attributes:**
   - For `Student`: `Roll_No` (Primary Key), `Name`, `Age`
   - For `Course`: `Course_ID` (Primary Key), `Course_Name`
3. **Relationships:**
   - `Enrolls` (between `Student` and `Course`)

2.2 Mapping Cardinality

Methodology:

Determining Cardinality Ratios Cardinality specifies the maximum number of relationship instances that an entity can participate in.

1. **One-to-One (1:1):** An entity in A is associated with at most one entity in B, and vice versa.
2. **One-to-Many (1:N):** An entity in A can be associated with many entities in B, but an entity in B is associated with at most one entity in A.
3. **Many-to-One (N:1):** Many entities in A are associated with at most one entity in B, and an entity in B can be associated with many entities in A.
4. **Many-to-Many (M:N):** An entity in A is associated with many entities in B, and an entity in B is associated with many entities in A.

Worked Example:

Analyzing Department and Employee **Problem:** In a company, a department employs multiple employees, but each employee works for exactly one department. A department has one manager, and an employee can manage at most one department. Determine the mapping cardinalities for these relationships.
Solution:

1. **Relationship `Works_For`:** Between `Employee` and `Department`. One department has many employees. One employee works for exactly one department. **Cardinality:** N:1 (Many Employees to One Department).
2. **Relationship `Manages`:** Between `Employee` and `Department`. One department has one manager. One manager manages one department. **Cardinality:** 1:1 (One Employee to One Department).

2.3 ER Diagrams

Methodology:

Constructing ER Diagrams Follow these mapping rules to draw an ER Diagram from requirements:

1. Draw **Rectangles** to represent Entity Sets.
2. Draw **Ellipses** to represent Attributes. Underline the Primary Key attribute.
3. Draw **Diamonds** to represent Relationship Sets.
4. Connect attributes to entities, and entities to relationships using solid lines.
5. Add cardinality constraints on the lines connecting entities to relationships (e.g. 1, N, M).

Worked Example:

Drawing ER Diagram for a Library **Problem:** A library system has books and members. A book has a `Book_ID` and `Title`. A member has a `Member_ID` and `Name`. A member can borrow multiple books, but a specific physical book can be borrowed by at most one member at a time. Define the schema structure conceptually.
Solution:

1. **Entities:** `Member` (`Member_ID`, `Name`) and `Book` (`Book_ID`, `Title`).

2. **Relationship:** `Borrows` (between `Member` and `Book`).
3. **Cardinality:** Since one member can borrow many books, but a physical book is with at most one member, the relationship is 1:N from `Member` to `Book`.
4. **Diagram Layout (Conceptual):**
   - `Member` [Rectangle] connected to `Member_ID` (Underlined) and `Name` [Ellipses].
   - `Book` [Rectangle] connected to `Book_ID` (Underlined) and `Title` [Ellipses].
   - `Borrows` [Diamond] connects `Member` and `Book`.
   - Line from `Member` to `Borrows` has a '1'. Line from `Borrows` to `Book` has an 'N'.

2.4 Weak Entity Sets

Methodology:

Handling Weak Entity Sets A weak entity set does not have a primary key of its own and depends on a strong entity set (owner) for its existence.

1. Identify the weak entity (entity that cannot exist without another entity).
2. Identify the identifying relationship (relationship connecting the weak entity to the owner).
3. Represent the weak entity with a **Double Rectangle**.
4. Represent the identifying relationship with a **Double Diamond**.
5. Identify the partial key (discriminator) of the weak entity and underline it with a **Dashed Line**.

Worked Example:

Employee and Dependents **Problem:** An employee has an `Emp_ID` and `Name`. Employees can have dependents (e.g. spouse or children). A dependent is uniquely identified by their `Dep_Name` only in the context of the Employee they belong to. Identify the weak entity set and its components.
Solution:

1. **Strong Entity:** `Employee` (`Emp_ID` is Primary Key).
2. **Weak Entity:** `Dependent`. It cannot exist without the `Employee`. Represented by a Double Rectangle.
3. **Partial Key:** `Dep_Name`. (Underlined with a dashed line).
4. **Identifying Relationship:** `Has_Dependent`. Represented by a Double Diamond.
5. **Total Participation:** The line between `Dependent` and `Has_Dependent` is a double line, indicating that every dependent must be linked to an employee.

2.5 Enhanced ER Model (Subclass, Super Class, Generalization, Specialization, Aggregation)

Methodology:

Applying Generalization and Specialization Generalization combines lower-level entities into a higher-level entity (bottom-up). Specialization breaks a higher-level entity into lower-level sub-entities (top-down).

1. Identify the **Superclass** (contains common attributes shared by all instances).

2. Identify the **Subclasses** (contains specific attributes unique to certain instances).
3. Draw a triangle labeled **ISA** (is-a) pointing to the superclass.
4. Connect subclasses to the ISA triangle.

Worked Example:

Bank Account Specialization **Problem:** A bank offers two types of accounts: Savings Account and Current Account. All accounts have an `Account_No` and `Balance`. Savings accounts have an `Interest_Rate`, while Current accounts have an `Overdraft_Limit`. Design the EER structure conceptually.
Solution:

1. **Superclass:** `Account` with attributes `Account_No` (Primary Key) and `Balance`.
2. **Subclasses:**
   - `Savings_Account` with specific attribute `Interest_Rate`.
   - `Current_Account` with specific attribute `Overdraft_Limit`.
3. **Relationship:** Both subclasses connect to an `ISA` triangle which points up to the `Account` superclass, indicating a specialization relationship.

Methodology:

Applying Aggregation Aggregation is an abstraction where a relationship between two entities is treated as a single entity, allowing it to participate in another relationship.

1. Identify the core relationship between two distinct entities.
2. Enclose the entities and their relationship in a large bounding box.
3. Treat this bounding box as a new composite entity.
4. Connect this composite entity to another relationship to avoid complex ternary relationships.

Worked Example:

Project Employee and Machinery **Problem:** Employees work on Projects. The assignment of a specific Employee to a specific Project requires particular Machinery. Model this using aggregation.
Solution:

1. **Core Relationship:** `Works_On` between `Employee` and `Project`.
2. **Aggregation Box:** Draw a box around `Employee`, `Project`, and `Works_On`. This creates a composite entity representing an employee's assignment to a project.
3. **Higher-level Relationship:** `Requires` connects the Aggregation Box to the `Machinery` entity.
4. **Interpretation:** Machinery is not assigned to an employee or a project independently; it is assigned to the specific occurrence of an employee working on a specific project.

2.6 Converting ER Diagrams to database

Methodology:

Algorithm for Converting ER to Relational Tables Convert ER models to a Relational Schema (Tables) using the following systematic rules:

1. **Strong Entities:** Create a table for each strong entity. The primary key of the entity becomes the primary key of the table.
2. **Weak Entities:** Create a table with the weak entity's attributes. Add the primary key of the owner entity as a foreign key. The primary key of this new table is the composite of the owner's primary key and the weak entity's partial key.
3. **1:1 Relationship:** Add the primary key of one table as a foreign key in the other table. If one side has total participation, always put the foreign key on that side.
4. **1:N Relationship:** Add the primary key of the "1" side as a foreign key in the table on the "N" side.
5. **M:N Relationship:** Create a new table. The primary key of this table is the combination of the primary keys of the participating entities. Any descriptive attributes of the relationship also go into this table.
6. **Multi-valued Attributes:** Create a new table containing the entity's primary key and the multi-valued attribute. Both form the composite primary key of the new table.

Worked Example:

Converting E Commerce ER to Tables **Problem:** Consider an ER diagram with the following details:

- Entity `Customer` (`Cust_ID`, `Name`, `Phone` [Multi-valued])
- Entity `Order` (`Order_ID`, `Date`)
- Relationship `Places` (1 Customer places N Orders)
- Entity `Product` (`Prod_ID`, `Price`)
- Relationship `Contains` (M Orders contain N Products)

Convert this ER diagram into a relational database schema.
Solution: Apply the conversion rules step-by-step:
Step 1: Strong Entities

- `Customer` table: `Cust_ID` (Primary Key), `Name`.
- `Order` table: `Order_ID` (Primary Key), `Date`.
- `Product` table: `Prod_ID` (Primary Key), `Price`.

Step 2: Multi-valued Attribute

- Customer has a multi-valued attribute `Phone`.
- Create table `Customer_Phone`: `Cust_ID` (Foreign Key), `Phone_Number`.
- Primary Key: (`Cust_ID`, `Phone_Number`).

Step 3: 1:N Relationship (Places)

- `Customer` (1) to `Order` (N).
- Add the primary key of the "1" side (`Cust_ID`) into the "N" side table (`Order`).

- Updated `Order` table: `Order_ID` (PK), `Date`, `Cust_ID` (FK).

Step 4: M:N Relationship (Contains)

- `Order` (M) to `Product` (N).
- Create a new table `Order_Details` for the relationship.
- Attributes: `Order_ID` (FK), `Prod_ID` (FK).
- Primary Key: (`Order_ID`, `Prod_ID`).

Final Relational Schema:

Table Name	Attributes (Primary Keys in bold)
Customer	**Cust_ID**, Name
Customer_Phone	**Cust_ID**, **Phone_Number**
Order	**Order_ID**, Date, Cust_ID (FK)
Product	**Prod_ID**, Price
Order_Details	**Order_ID**, **Prod_ID**

CHAPTER 3
Structured Query Language

3.1 Introduction and Basic Commands

3.1.1 Data Types

Methodology:

Common SQL Data Types SQL requires data types to be defined for each column. The most frequently used types include:

1. **INT**: Used for storing whole numbers.
2. **VARCHAR(n)**: Variable-length character string of maximum length n.
3. **CHAR(n)**: Fixed-length character string of length n.
4. **DECIMAL(p s)**: Numeric data where p is the total number of digits and s is the number of digits after the decimal point.
5. **DATE**: Stores calendar dates in Year-Month-Day format.

3.1.2 Data Definition Language DDL

Methodology:

DDL Commands Data Definition Language (DDL) is used to create and modify the structure of database objects.

1. **CREATE**: Constructs a new table view or database.
2. **ALTER**: Modifies an existing database structure.
3. **DROP**: Permanently deletes a table and its structure from the database.
4. **TRUNCATE**: Deletes all data inside a table but retains the empty table structure.

Worked Example:

Creating and Altering Tables **Problem:** Write SQL commands to create a `Student` table with `ID` (Integer) and `Name` (VARCHAR(50)). Then add a `City` column and remove the `Name` column.
Solution:

1. Create the table:
   `CREATE TABLE Student (ID INT, Name VARCHAR(50));`
2. Add the new column:
   `ALTER TABLE Student ADD City VARCHAR(50);`

3. Drop the existing column:
   `ALTER TABLE Student DROP COLUMN Name;`

3.1.3 Data Manipulation Language DML

Methodology:

DML Commands Data Manipulation Language (DML) manages data within existing tables.

1. **INSERT INTO**: Adds new rows to a table.
2. **UPDATE**: Modifies existing rows based on a specific condition.
3. **DELETE**: Removes specific rows based on a condition.

Worked Example:

Inserting and Updating Data **Problem:** Write SQL commands to insert a student (ID: 101, City: 'Surat') into the `Student` table and then update their city to 'Ahmedabad'.
Solution:

1. Insert the record:
   `INSERT INTO Student (ID, City) VALUES (101, 'Surat');`
2. Update the record:
   `UPDATE Student SET City = 'Ahmedabad' WHERE ID = 101;`

3.1.4 Data Query Language DQL

Methodology:

DQL Command Data Query Language (DQL) consists of a single powerful command to retrieve data.

1. **SELECT**: Fetches data from one or more tables. The basic syntax is:
   `SELECT column1, column2 FROM table_name WHERE condition;`

3.1.5 Privilege Commands DCL

Methodology:

Data Control Language DCL Data Control Language (DCL) commands manage database access rights and security.

1. **GRANT**: Gives specific privileges to a user.
2. **REVOKE**: Removes specific privileges from a user.

3.1.6 Miscellaneous Commands TCL

Methodology:

Transaction Control Language TCL Transaction Control Language (TCL) manages transactions to maintain database consistency.

1. **COMMIT**: Saves all transactions made since the last commit permanently to the database.
2. **ROLLBACK**: Undoes transactions not yet saved.

3. **SAVEPOINT**: Creates a point within a transaction to which you can later roll back.

3.2 SQL Views

Methodology:

Creating and Using Views A View is a virtual table whose contents are defined by an underlying SQL query. Views hide query complexity and enhance data security.

1. **Create a View**:
   `CREATE VIEW view_name AS SELECT columns FROM table WHERE condition;`
2. **Query a View**: Treat the view like a regular table.
   `SELECT * FROM view_name;`
3. **Drop a View**:
   `DROP VIEW view_name;`

Worked Example:

Implementing a Security View **Problem:** An `Employee` table contains `EmpID`, `Name`, `Department`, and `Salary`. Create a view named `PublicEmpView` that hides the `Salary` column. Then write a query to read from this view.
Solution:

1. Create the view including only non-sensitive columns:
   ```
   CREATE VIEW PublicEmpView AS
   SELECT EmpID, Name, Department FROM Employee;
   ```
2. Retrieve data from the newly created view:
   `SELECT * FROM PublicEmpView;`

3.3 SQL Functions

Methodology:

Aggregate and Scalar Functions SQL provides built-in functions to perform calculations on data.

1. **Aggregate Functions**: Return a single value from a group of values.
   - `COUNT()`: Returns the number of rows.
   - `SUM()`: Returns the total sum of a numeric column.
   - `AVG()`: Returns the average value.
   - `MAX() / MIN()`: Returns the highest or lowest value respectively.
2. **Scalar Functions**: Return a single value based on the input value (e.g. `UPPER()`, `LOWER()`, `LENGTH()`).

Worked Example:

Applying SQL Aggregate Functions **Problem:** Given a table `Sales` with columns `ProductID` and `Revenue`, calculate the total revenue, the average revenue, and the number of products sold.
Solution:

1. Total Revenue:
   `SELECT SUM(Revenue) FROM Sales;`

2. Average Revenue:
   `SELECT AVG(Revenue) FROM Sales;`
3. Total Number of Products:
   `SELECT COUNT(ProductID) FROM Sales;`

3.4 SQL Operators

Methodology:

Relational and Logical Operators Operators are used within the `WHERE` clause to filter records.

1. **Arithmetic**: `+`, `-`, `*`, `/`
2. **Comparison**: `=`, `<>` or `!=`, `>`, `<`, `>=`, `<=`
3. **Logical**: `AND`, `OR`, `NOT`
4. **Special Operators**:
   - `BETWEEN a AND b`: Inclusive range filtering.
   - `IN (val1, val2)`: Matches any value in a list.
   - `LIKE`: Pattern matching using wildcards (`%` for zero or more characters, `_` for a single character).

Worked Example:

Filtering Data with Special Operators **Problem:** Write SQL queries to find employees who: 1. Have salaries between 50000 and 80000. 2. Work in the 'HR' or 'IT' departments. 3. Have names starting with the letter 'A'.
Solution:

1. Using `BETWEEN`:
   `SELECT * FROM Employee WHERE Salary BETWEEN 50000 AND 80000;`
2. Using `IN`:
   `SELECT * FROM Employee WHERE Department IN ('HR', 'IT');`
3. Using `LIKE`:
   `SELECT * FROM Employee WHERE Name LIKE 'A%';`

3.5 Set Operators

Methodology:

SQL Set Operations Set operators combine the result sets of two or more `SELECT` queries into a single result set.

1. **UNION**: Combines results and removes duplicate rows.
2. **UNION ALL**: Combines results but keeps duplicate rows.
3. **INTERSECT**: Returns only rows that appear in both result sets.
4. **MINUS (or EXCEPT)**: Returns rows from the first query that are not present in the second query.

Worked Example:

Applying Set Operations **Problem:** Consider two tables `QueryA` (cities in India) and `QueryB` (cities with population > 10 million). Find all unique cities from both lists, and then find cities that are in both lists.
Solution:

1. Combine unique cities using `UNION`:
```
SELECT CityName FROM QueryA
UNION
SELECT CityName FROM QueryB;
```
2. Find overlapping cities using `INTERSECT`:
```
SELECT CityName FROM QueryA
INTERSECT
SELECT CityName FROM QueryB;
```

3.6 Joins

Methodology:

Joining Multiple Tables Joins are used to combine rows from two or more tables based on a related column between them.

1. **INNER JOIN**: Returns records that have matching values in both tables.
2. **LEFT JOIN**: Returns all records from the left table and matched records from the right table. Missing matches contain NULL.
3. **RIGHT JOIN**: Returns all records from the right table and matched records from the left table.
4. **FULL OUTER JOIN**: Returns all records when there is a match in either left or right table.

Worked Example:

Writing JOIN Queries **Problem:** Given tables `Students(StudentID, Name)` and `Grades(StudentID, Course, Grade)`. Retrieve a list of all students and their grades. If a student has no grades their name should still appear in the result.
Solution: Since we need all students regardless of whether they have a grade we must use a `LEFT JOIN`.

```
SELECT Students.Name, Grades.Course, Grades.Grade
FROM Students
LEFT JOIN Grades ON Students.StudentID = Grades.StudentID;
```

3.7 SQL Constraints

Methodology:

Integrity Constraints Constraints restrict the type of data that can be inserted into a table, ensuring data accuracy and reliability.

1. **Domain Integrity**: Validates entries for a specific column.
   - **NOT NULL**: Ensures a column cannot have a NULL value.
   - **CHECK**: Ensures all values in a column satisfy a specific condition.
   - **DEFAULT**: Sets a default value for a column if no value is specified.
2. **Entity Integrity**: Ensures each row in a table is uniquely identifiable.

- **PRIMARY KEY**: Uniquely identifies each record. Cannot be NULL.
- **UNIQUE**: Ensures all values in a column are different.

3. **Referential Integrity**: Ensures relationships between tables remain valid.
   - **FOREIGN KEY**: A key used to link two tables together by pointing to a `PRIMARY KEY` in another table.

Worked Example:

Implementing Table Constraints **Problem:** Create an `Orders` table enforcing the following rules: 1. `OrderID` must uniquely identify the row (Entity Integrity). 2. `Quantity` must be strictly greater than 0 (Domain Integrity). 3. `CustomerID` must link to a valid customer in the `Customers` table (Referential Integrity).
Solution:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    Quantity INT CHECK (Quantity > 0),
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

CHAPTER 4

Normalization

4.1 Anomalies Created by Poor Database Design

Poorly designed databases suffer from data redundancy, leading to data inconsistency and maintenance issues known as anomalies. Consider an unnormalized relation with attributes: StudentID, StudentName, CourseID, CourseFee.

1. **Insertion Anomaly:** Occurs when certain attributes cannot be inserted into the database without the presence of other attributes. Example: We cannot add a new course and its fee until a student enrolls in it because StudentID might be part of the primary key.
2. **Update Anomaly:** Occurs when data redundancy requires multiple updates for a single logical change. Example: If a CourseFee changes, we must update every row where a student is taking that course. Missing a row leads to inconsistent data.
3. **Deletion Anomaly:** Occurs when deleting a row inadvertently causes the loss of other unrelated data. Example: If the only student enrolled in a course drops out and we delete their record, we also lose all information about the course and its fee.

4.2 Normalization Definition and Goals

Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update, and Deletion anomalies.

Goals of Normalization:

- Eliminate redundant data (storing the same data in more than one table).
- Ensure data dependencies make sense (only storing related data in a table).
- Reduce the amount of space a database consumes.
- Ensure logical data consistency.

4.3 Functional Dependencies

A Functional Dependency (FD) is a relationship that exists when one attribute uniquely determines another attribute. If R is a relation and X and Y are sets of attributes in R, then $X \to Y$ (read as "X determines Y") means that for any two tuples, if their X values are identical, their Y values must also be identical.

Methodology:

Analyzing Functional Dependencies

1. **Trivial Functional Dependency:** $X \to Y$ is trivial if Y is a subset of X. Example: $\{A, B\} \to B$.
2. **Non-Trivial Functional Dependency:** $X \to Y$ is non-trivial if Y is not a subset of X. Example: $A \to B$.

3. **Fully Functional Dependency:** In $X \to Y$, Y is fully functionally dependent on X if removing any attribute from X means the dependency no longer holds.
4. **Partial Dependency:** Occurs when a non-prime attribute is functionally dependent on only a part of a composite primary key.
5. **Transitive Dependency:** Occurs when a non-prime attribute depends on another non-prime attribute ($A \to B$ and $B \to C$ means $A \to C$).

4.4 Normal Forms

Normalization proceeds in stages called Normal Forms. Each normal form imposes stricter rules on the relation structure to resolve specific anomalies.

4.4.1 First Normal Form (1NF)

Methodology:

Conversion to First Normal Form 1NF A relation is in 1NF if and only if every attribute contains atomic (indivisible) values. No multi-valued attributes or repeating groups are allowed. **Steps to Convert to 1NF:**

1. Identify any multi-valued attributes in the table.
2. Create a separate row for each value in the multi-valued attribute, duplicating the other non-repeating data for that entity.
3. Ensure a primary key exists that uniquely identifies each newly created row (usually expanding to a composite key).

Worked Example:

Converting an Unnormalized Table to 1NF **Problem:** Convert the following Student table into 1NF.

StudentID	Name	PhoneNumbers
1	Alice	9999911111, 8888822222
2	Bob	7777733333

Solution:
Step 1: Identify multi-valued attributes. The `PhoneNumbers` column contains multiple values for StudentID 1.
Step 2: Create separate rows for multi-valued data. Split Alice's phone numbers into two distinct rows. Duplicate her StudentID and Name.

StudentID	Name	PhoneNumber
1	Alice	9999911111
1	Alice	8888822222
2	Bob	7777733333

Step 3: Define the new Primary Key. The new primary key is the composite key (StudentID, PhoneNumber). The table is now in 1NF.

4.4.2 Second Normal Form (2NF)

Methodology:

Conversion to Second Normal Form 2NF A relation is in 2NF if it is in 1NF and contains no **partial dependencies**. This means every non-prime attribute must be fully functionally dependent on the entire primary key. **Steps to Convert to 2NF:**

1. Ensure the table is in 1NF.
2. Identify the Primary Key. If it's a single attribute, the table is automatically in 2NF.
3. If the Primary Key is composite, identify any non-prime attributes that depend on only a part of the primary key.
4. Remove the partially dependent attributes to a new table along with a copy of the part of the primary key they depend on.

Worked Example:

Eliminating Partial Dependencies for 2NF **Problem:** Consider the `Enrollment` table with Primary Key (StudentID, CourseID). Convert it to 2NF.

StudentID	CourseID	CourseName	Grade
1	C101	DBMS	A
2	C101	DBMS	B
1	C102	Java	A

Solution:
Step 1: Check 1NF and Primary Key. Table is in 1NF. Primary Key is the composite set (StudentID, CourseID).
Step 2: Identify partial dependencies.

- `Grade` depends on both StudentID and CourseID – Fully dependent.
- `CourseName` depends ONLY on `CourseID` – **Partial dependency**.

Step 3: Decompose the table to remove partial dependencies. Create one table for the full dependency and another for the partial dependency.
Table 1: EnrollmentGrade (in 2NF)
Primary Key: (StudentID, CourseID)

StudentID	CourseID	Grade
1	C101	A
2	C101	B
1	C102	A

Table 2: Course (in 2NF)
Primary Key: CourseID

CourseID	CourseName
C101	DBMS
C102	Java

Both tables are now free of partial dependencies and are in 2NF.

4.4.3 Third Normal Form (3NF)

Methodology:

Conversion to Third Normal Form 3NF A relation is in 3NF if it is in 2NF and contains no **transitive dependencies**. A non-prime attribute must not depend on another non-prime attribute. **Steps to Convert to 3NF:**

1. Ensure the table is in 2NF.
2. Identify functional dependencies among non-prime attributes (e.g. $A \to B \to C$).
3. Remove the transitively dependent attribute(s) to a new table.
4. Copy the determinant attribute (the non-prime attribute that determines the others) into the new table to serve as its primary key.

Worked Example:

Eliminating Transitive Dependencies for 3NF **Problem:** Convert the `StudentInfo` table to 3NF. Primary Key is StudentID.

StudentID	Name	ZipCode	City
1	Alice	380001	Ahmedabad
2	Bob	380002	Ahmedabad
3	Charlie	390001	Vadodara

Solution:
Step 1: Check 2NF and Primary Key. Table is in 2NF (single attribute primary key means no partial dependency). Primary Key is StudentID.
Step 2: Identify transitive dependencies.

- `StudentID` → `ZipCode`
- `ZipCode` → `City`

Because `City` depends on `ZipCode` (which is a non-prime attribute), this is a transitive dependency: `StudentID` → `ZipCode` → `City`.
Step 3: Decompose the table. Separate the transitively dependent relationship into a new table.
Table 1: Student (in 3NF)
Primary Key: StudentID

StudentID	Name	ZipCode
1	Alice	380001
2	Bob	380002
3	Charlie	390001

Table 2: Location (in 3NF)
Primary Key: ZipCode

ZipCode	City
380001	Ahmedabad
380002	Ahmedabad
390001	Vadodara

Both tables are now in 3NF.

4.4.4 Boyce-Codd Normal Form (BCNF)

Methodology:

Conversion to Boyce Codd Normal Form BCNF BCNF is a stronger version of 3NF. A relation is in BCNF if for every non-trivial functional dependency $X \to Y$, X is a superkey. In simpler terms, a prime attribute cannot depend on a non-prime attribute. **Steps to Convert to BCNF:**

1. Ensure the table is in 3NF.
2. Find all functional dependencies. If there is an FD $A \to B$ where A is not a candidate key (or superkey), the table violates BCNF.
3. Decompose the table: Create a new table with attributes A and B.
4. Remove B from the original table, leaving A as a foreign key linking to the new table.

Worked Example:

Converting a 3NF Relation to BCNF **Problem:** Consider the `StudentProject` table. A student can take multiple subjects, but a subject is taught by only one professor. A professor teaches only one subject. Candidate Key: (StudentID, Subject).

StudentID	Subject	Professor
1	Database	Dr. Smith
2	Database	Dr. Smith
1	Java	Dr. Jones

Solution:
Step 1: Check 3NF. There are no partial dependencies (`Professor` depends on the combination of StudentID and Subject, not just one) and no transitive dependencies. The table is in 3NF.
Step 2: Check for BCNF violations. The functional dependencies are:

- (StudentID, Subject) → Professor
- Professor → Subject (Because each professor teaches only one subject).

In the dependency `Professor` → `Subject`, `Professor` is not a candidate key. Therefore, the relation is not in BCNF.
Step 3: Decompose the table. Separate the violating dependency.
Table 1: ProfessorSubject (in BCNF)
Primary Key: Professor

Professor	Subject
Dr. Smith	Database
Dr. Jones	Java

Table 2: StudentProfessor (in BCNF)
Primary Key: (StudentID, Professor)

StudentID	Professor
1	Dr. Smith
2	Dr. Smith
1	Dr. Jones

Both tables are now in BCNF because the determinant of every functional dependency is a candidate key.

CHAPTER 5

Transaction Management

5.1 Basic Transaction Concepts

A transaction is a logical unit of work that contains one or more SQL statements. A transaction must either complete entirely or not at all, ensuring data consistency in the database.

Methodology:

Understanding ACID Properties Every transaction must satisfy the four essential ACID properties to maintain database integrity:

1. **Atomicity:** The "all or nothing" rule. Either all operations of the transaction are executed, or none are.
2. **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
3. **Isolation:** Multiple transactions executing concurrently must yield the same results as if they executed serially.
4. **Durability:** Once a transaction commits, its changes are permanent and survive future system and database failures.

Methodology:

Transaction State Transitions A transaction transitions through several states during its lifecycle:

1. **Active:** The initial state. The transaction stays in this state while executing.
2. **Partially Committed:** After the final statement has been executed, but before changes are permanently saved.
3. **Failed:** When normal execution can no longer proceed due to an error.
4. **Aborted:** After the transaction has been rolled back and the database restored to its prior state.
5. **Committed:** After successful completion, where changes are saved to the transaction log and database.

Worked Example:

Tracing Transaction States Consider a transaction T_1 that transfers Rs 500 from Account A to Account B. Trace its states under a scenario where the system crashes immediately after deducting from A but before adding to B.

Solution:

1. **Active:** T_1 starts execution. Reads Account A balance.
2. **Active:** T_1 deducts 500 from Account A.
3. **Failed:** System crashes before T_1 can read and update Account B. The transaction cannot proceed.

4. **Aborted:** Upon system recovery, the DBMS detects an incomplete transaction T_1 in the transaction log. It rolls back the deduction from Account A to restore consistency.

5.2 Concurrency Control

Concurrency control manages simultaneous execution of multiple transactions in a multiprogramming database environment. Without it, concurrent execution can lead to severe data anomalies.

Methodology:

Identifying Concurrency Anomalies To analyze concurrent schedules, look for the following common problems:

1. **Lost Update Problem (Write-Write Conflict):** Occurs when two transactions read the same data and update it concurrently. The second update overwrites the first.
2. **Dirty Read Problem (Write-Read Conflict):** Occurs when a transaction reads data that has been updated by an uncommitted transaction. If the uncommitted transaction fails, the first transaction has read invalid data.
3. **Unrepeatable Read Problem (Read-Write Conflict):** Occurs when a transaction reads the same data twice and gets different values because another transaction modified the data in between.

Worked Example:

Detecting the Dirty Read Problem Given the following schedule of transactions T_1 and T_2 operating on data item X (initial value $X = 100$), identify the anomaly.

Transaction T_1	**Transaction T_2**
1. Read(X)	
2. X = X + 50	
3. Write(X)	
4.	Read(X)
5.	X = X * 2
6.	Write(X)
7.	Commit
8. Rollback	

Solution:

1. T_1 reads $X = 100$ and updates it to 150. It writes $X = 150$ to the buffer.
2. T_2 reads $X = 150$ (dirty data) written by T_1.
3. T_2 multiplies by 2, writes $X = 300$, and Commits.
4. T_1 fails at step 8 and rolls back. The original value $X = 100$ should be restored.
5. However, T_2 has already committed using the temporary value 150. This is a **Dirty Read Problem** because T_2 read uncommitted data from T_1.

5.3 Serializability of Transactions

A schedule is serializable if its outcome is equivalent to the outcome of its transactions executing serially (one after another) in some order. Conflict serializability is the most common method to guarantee correctness.

Methodology:

Testing Conflict Serializability using Precedence Graph Follow these steps to determine if a schedule is conflict serializable:

1. **Identify Transactions:** Create a node (vertex) for each transaction T_i in the schedule.
2. **Identify Conflicts:** Find all pairs of conflicting operations. Two operations conflict if:
   - They belong to different transactions.
   - They access the same data item.
   - At least one of them is a Write operation (i.e. Read-Write, Write-Read, Write-Write).
3. **Draw Edges:** For every conflict where operation in T_i executes before operation in T_j, draw a directed edge from $T_i \to T_j$.
4. **Cycle Detection:** Check the resulting precedence graph.
   - If the graph contains **no cycles**, the schedule is **Conflict Serializable**. The serial order is found via topological sorting.
   - If the graph contains a **cycle**, the schedule is **Not Conflict Serializable**.

Worked Example:

Testing Schedule for Conflict Serializability Determine whether the following schedule S involving transactions T_1, T_2, and T_3 is conflict serializable.
S: $R_1(A)$, $W_2(A)$, $Commit_2$, $W_1(A)$, $Commit_1$, $W_3(A)$, $Commit_3$
Solution:

1. **Identify Nodes:** Create nodes for T_1, T_2, and T_3.
2. **Identify Conflicts and Draw Edges:**
   - $R_1(A)$ in T_1 and $W_2(A)$ in T_2: Conflict on A. T_1 reads before T_2 writes. Draw edge $T_1 \to T_2$.
   - $R_1(A)$ in T_1 and $W_3(A)$ in T_3: Conflict on A. T_1 reads before T_3 writes. Draw edge $T_1 \to T_3$.
   - $W_2(A)$ in T_2 and $W_1(A)$ in T_1: Conflict on A. T_2 writes before T_1 writes. Draw edge $T_2 \to T_1$.
   - $W_2(A)$ in T_2 and $W_3(A)$ in T_3: Conflict on A. T_2 writes before T_3 writes. Draw edge $T_2 \to T_3$.
   - $W_1(A)$ in T_1 and $W_3(A)$ in T_3: Conflict on A. T_1 writes before T_3 writes. Draw edge $T_1 \to T_3$.
3. **Analyze Graph:** The graph has edges $T_1 \to T_2$ and $T_2 \to T_1$.
4. **Conclusion:** Because there is a cycle ($T_1 \leftrightarrow T_2$), the schedule S is **Not Conflict Serializable**.

Worked Example:

Finding Serial Schedule Order Check if schedule S is conflict serializable. If yes find the equivalent serial schedule.
S: $R_1(A)$, $R_2(A)$, $W_1(A)$, $R_3(A)$, $W_2(B)$, $W_3(A)$
Solution:

1. **Nodes:** T_1, T_2, T_3.
2. **Conflicts on Data A:**
   - $R_2(A)$ and $W_1(A)$: T_2 reads before T_1 writes $\to$ Edge $T_2 \to T_1$
   - $W_1(A)$ and $R_3(A)$: T_1 writes before T_3 reads $\to$ Edge $T_1 \to T_3$
   - $W_1(A)$ and $W_3(A)$: T_1 writes before T_3 writes $\to$ Edge $T_1 \to T_3$

- $R_2(A)$ and $W_3(A)$: T_2 reads before T_3 writes $\rightarrow$ Edge $T_2 \rightarrow T_3$
- $R_1(A)$ and $W_3(A)$: T_1 reads before T_3 writes $\rightarrow$ Edge $T_1 \rightarrow T_3$

3. **Conflicts on Data B:** Only accessed by T_2. No conflict.
4. **Precedence Graph Edges:** $T_2 \rightarrow T_1$, $T_1 \rightarrow T_3$, $T_2 \rightarrow T_3$.
5. **Cycle Detection:** There is no cycle in the graph. Thus, the schedule is **Conflict Serializable**.
6. **Equivalent Serial Order:** Perform topological sorting. T_2 has no incoming edges. After removing T_2, T_1 has no incoming edges. Finally T_3. The equivalent serial schedule is $T_2 \rightarrow T_1 \rightarrow T_3$.

5.4 Locking Methods for Concurrency Control

Locking protocols ensure isolation by requiring transactions to obtain locks on data items before accessing them.

Methodology:

Basic Locking Operations

1. **Shared Lock (S):** Allows a transaction to read a data item but not update it. Multiple transactions can hold Shared locks on the same item.
2. **Exclusive Lock (X):** Allows a transaction to both read and update a data item. Only one transaction can hold an Exclusive lock on an item at a time.

Methodology:

Two Phase Locking Protocol The Two-Phase Locking (2PL) protocol ensures conflict serializability by requiring that every transaction acquires and releases locks in two distinct phases:

1. **Growing Phase:** A transaction may obtain new locks on data items but cannot release any lock.
2. **Lock Point:** The point where the transaction has acquired all its needed locks.
3. **Shrinking Phase:** A transaction may release existing locks but cannot obtain any new locks.

Note: Strict 2PL requires that all Exclusive locks be held until the transaction commits or aborts to prevent dirty reads.

Worked Example:

Evaluating Schedule against Two Phase Locking Determine if the following transaction T_1 follows the basic Two-Phase Locking protocol.
T_1: Lock-S(A), Read(A), Lock-X(B), Read(B), Write(B), Unlock(A), Lock-X(C), Read(C), Write(C), Unlock(B), Unlock(C).
Solution:

1. **Analyze Lock Operations:**
   - Acquires Lock-S(A)
   - Acquires Lock-X(B)
   - **Releases** Unlock(A) — *Shrinking phase begins here.*
   - **Acquires** Lock-X(C) — *Violation!* A new lock is acquired after a lock has been released.
2. **Conclusion:** The transaction T_1 **does not** follow the Two-Phase Locking protocol because it requests a new lock (Lock-X(C)) after it has already released a lock (Unlock(A)).

Worked Example:

Transforming to Two Phase Locking Modify transaction T_1 from the previous example so that it satisfies the Two-Phase Locking protocol.
Solution: To satisfy 2PL, all lock acquisitions must occur before any lock release. We delay the Unlock(A) until all locks are acquired.
Modified T_1:

1. Lock-S(A), Read(A)
2. Lock-X(B), Read(B), Write(B)
3. **Lock-X(C)** — *All locks are now acquired (Lock Point)*
4. **Unlock(A)** — *Shrinking phase begins*
5. Read(C), Write(C)
6. Unlock(B)
7. Unlock(C)

This modified sequence follows 2PL because no new lock is requested after the first Unlock operation.

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

General Aggregations (Unit 1)

In many foundational database problems, such as calculating storage requirements, query statistics, or evaluating student marks examples, aggregate totals and percentages are commonly required.

Total Sum Calculation: The aggregate sum of individual components is formulated as:

$$\text{Total} = \sum_{i=1}^{n} v_i$$

where v_i represents the value of each individual component.

Percentage Calculation: To find the relative percentage of an obtained value against a maximum capacity or total:

$$\text{Percentage} = \left(\frac{\text{Obtained Value}}{\text{Maximum Possible Value}} \right) \times 100$$

Transaction Management (Unit 5)

During concurrent transaction executions, database items (such as X) are continuously read and modified. The formulas below abstract the operations applied to these items.

State Transition (Update Operation): A transaction frequently updates a database item based on its previous state:

$$X_{new} = X_{old} \pm \Delta$$

where Δ is the change applied by the transaction.

Direct Assignment (Blind Write): In some schedules, a transaction simply overwrites the database item with a new constant value without reading it first:

$$X = C$$

where C is a specific constant value.