Textbook for DI03016041

Theories & Fundamentals
Subject Code: DI03016041

Milav Dabgar

July 1, 2026

Textbook for DI03016041

First Edition: 2026

Author: Milav Dabgar
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.

Contents

List of Figures

List of Tables

Preface

The true power of the internet is not in connecting computers, but in connecting the physical world around us.

About this Book

This textbook is meticulously designed to align with the **GTU Diploma Syllabus (4353201)** for Wireless Sensor Networks and IoT. It provides a robust, intuitive, and comprehensive foundation in the rapidly evolving fields of sensor technologies, embedded systems, and the Internet of Things. Bridging the gap between theoretical network protocols and practical hardware implementations, this book decodes complex architectures into accessible, real-world analogies and engaging visual diagrams.

Target Audience

This book is specifically crafted for Diploma engineering students in the Information and Communication Technology (ICT) branch, as well as allied fields. It serves as an essential guide to demystifying the complexities of hardware-software integration, empowering students to build and understand the next generation of smart infrastructure.

Scope and Coverage

The coverage is strategically divided into the five core units of the official syllabus:

- **Unit 1: Introduction to WSN** – Exploring the fundamental building blocks of sensor nodes, network topologies, and the critical design principles prioritizing energy efficiency.
- **Unit 2: MAC and Routing Protocols** – Navigating the intricate layers of communication, addressing collision avoidance, duty cycling, and intelligent geographic data routing.
- **Unit 3: Overview of IoT** – Understanding the conceptual framework, reference architectures, and the foundational technologies (RFID, Big Data, Cloud) driving the IoT revolution.
- **Unit 4: Architecture and Implementation of IoT** – A deep dive into practical implementation, covering sensor integration, standard protocols (MQTT, CoAP), prototyping boards (NodeMCU, Raspberry Pi), and crucial security challenges.
- **Unit 5: IoT Applications** – Exploring transformative real-world use cases, from Smart Parking and Healthcare monitoring to Smart City infrastructure.

Milav Dabgar

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.

Milav Dabgar

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.

CHAPTER 1

Introduction to Database Systems

1.1 Introduction to Data, Databases, and DBMS

1.1.1 Data vs. Information

In computer science, **Data** refers to raw, unorganized facts and figures that lack context (e.g., "98", "Smith", "101"). **Information** is data that has been processed, structured, and presented in a given context to make it meaningful and useful (e.g., "Student ID 101, John Smith, scored 98 marks in Mathematics").

1.1.2 Database and Database Management System

A **Database (DB)** is a highly organized, logically coherent collection of related data stored electronically in a computer system. It is designed to allow easy retrieval, updating, and management of data. A **Database Management System (DBMS)** is a complex software application (like Oracle, MySQL, or SQL Server) that interacts with end-users, applications, and the database itself to capture and analyze data. The DBMS provides the interface to define, construct, manipulate, and share the database among various users and applications.

1.1.3 Purpose of Database System and Application

The primary purpose of a database system is to overcome the massive limitations of traditional file-processing systems (like storing data in text files or Excel spreadsheets). Databases provide: 1. Data isolation and security. 2. Concurrent access for multiple users (millions of people checking Amazon simultaneously). 3. Protection from hardware crashes (data recovery). 4. Efficient querying of massive datasets.

1.1.4 Basic Terminology

- **Metadata:** "Data about data." It is a directory that defines the structure of the database (e.g., "The 'Age' column must contain only integers between 0 and 120"). - **Data Dictionary:** A centralized repository within the DBMS that stores all the metadata. The DBMS uses it constantly to verify queries. - **Fields (Attributes):** The smallest unit of data, representing a column in a table (e.g., "First Name"). - **Records (Tuples):** A complete row of data in a table, containing all fields for one specific entity (e.g., the complete file on Student John). - **Data Items:** The actual, specific value stored at the intersection of a row and a column (e.g., "John").

1.2 File-Oriented System vs. Database System & Database Administrator

1.2.1 File-Oriented System vs. Database System

Before the invention of DBMS in the 1970s, companies used File-Oriented Systems (storing records in text files manipulated by C or COBOL programs). This caused catastrophic problems that DBMS was invented to solve.

1.2.2 The Database Administrator (DBA)

A DBMS is an incredibly complex, mission-critical piece of corporate infrastructure. The **Database Administrator (DBA)** is the senior IT professional responsible for the performance, integrity, and security of the database.

Roles and Responsibilities of a DBA: 1. **Schema Definition:** Designing the original structure (tables, relationships) of the database. 2. **Storage Structure and Access Method Definition:** Deciding exactly how the data is stored on the physical hard drives and creating indexes to make searches incredibly fast. 3. **Granting of Authorization:** Creating user accounts, assigning passwords, and determining exactly which users are allowed to see

Feature	File-Oriented System	Database System (DBMS)
Data Redundancy	High. The HR department and Payroll department maintain separate files, duplicating employee data.	Minimal. Data is stored centrally in one place and shared across all departments.
Data Inconsistency	If a user changes their address in the HR file, the Payroll file is not updated, causing conflicting data.	Because data is centralized, updating an address updates it universally for all applications.
Data Dependence	The structure of the data is hard-coded into the application software. If you change a data format from integer to float, the software breaks.	High Data Independence. The data structure is abstracted away from the application software.
Concurrent Access	Impossible. If two users try to edit a text file simultaneously, the file gets corrupted.	Handled gracefully through Transaction Management and locking protocols.

Table 1.1: File System vs. DBMS

which tables (Access Control). 4. **Routine Maintenance:** Performing daily backups, monitoring server CPU and RAM usage, and restoring the database if the server crashes.

1.3 Schema, Sub-Schema, and Instances

To understand how a DBMS organizes data, we must differentiate between the *design* of the database and the *actual data* inside it.

1.3.1 1. Schema (The Blueprint)

The **Schema** is the overall, permanent logical design of the entire database. It defines what tables exist, what columns are in those tables, the data types of those columns, and the relationships between the tables. - The schema is created once (usually by the DBA) when the database is first built. - It is analogous to the architectural blueprint of a house. It shows where the walls and rooms are, but does not tell you who is currently living in the rooms. - Because it represents the permanent structure, the schema very rarely changes.

1.3.2 2. Instance (The Snapshot)

An **Instance** is the actual, physical data stored in the database at one specific moment in time. - While the schema is permanent, the instance changes every single millisecond. - Every time a user clicks "Add to Cart" on Amazon, a new row is inserted, meaning the database has transitioned to a brand new instance. - Using the house analogy, the instance is a photograph of the people currently sitting inside the rooms today.

1.3.3 3. Sub-Schema (The User View)

A massive corporate database might have 5,000 tables. However, a specific user (like an entry-level HR clerk) should not be allowed to see the financial accounting tables or the CEO's salary. A **Sub-Schema** (often called a "View") is a customized, restricted subset of the overall schema presented to a specific user. - The HR clerk is given a sub-schema that only allows them to see the $`Employee_Name`and`Employee_Address`columns, effectively hiding the rest of the massive database from them. - Sub-schemas enforce strict security and greatly simplify the system for the end-user, as they only see the data relevant to their spec$

1.4 Data Abstraction and Data Independence

1.4.1 Data Abstraction (The 3-Tier Architecture)

A major purpose of a DBMS is to hide the brutal complexity of physical data storage from the user. This hiding is called **Data Abstraction**. The system is divided into three distinct levels of abstraction:

1. **Internal Level (Physical Level):** This is the lowest level of abstraction, closest to the hardware. It describes exactly *how* the data is physically stored on the spinning hard drives or SSDs. It deals with bytes, disk sectors, B-Tree indexes, and data compression. Only the DBA interacts with this level. 2. **Conceptual Level (Logical Level):** This is the middle level. It describes *what* data is stored in the entire database and what relationships exist between those data points. It views the data as clean, logical tables and rows (e.g., a "Student" table linked to a "Course" table),

completely ignoring the physical hard drives. Programmers and DBAs work at this level. 3. **External Level (View Level):** This is the highest level of abstraction, closest to the end-user. It only describes a *small portion* of the database (the Sub-Schema) relevant to a specific user. The user sees a graphical interface (like a website form) and is entirely unaware of the tables or the physical hard drives beneath them.

1.4.2 Data Independence

Because the DBMS uses this 3-tier architecture, it achieves **Data Independence**, which is the ability to modify the schema definition at one level without fundamentally breaking the schema at the higher level.

1. **Logical Data Independence:** The ability to modify the middle Conceptual schema without having to rewrite the application programs at the External level. *(Example: If the DBA adds a completely new column called "Blood_Type" to the Student table, the existing application software that prints student report cards will not crash; it will simply ignore the new column).* 2. **Physical Data Independence:** The ability to modify the lowest Internal schema without having to change the Conceptual or External schemas. *(Example: If the DBA unplugs the old HDD hard drives and migrates the database to brand new SSD hard drives, changing how data is physically written, the logical SQL tables and the user application software do not need to be changed at all. They won't even notice the hardware changed, except that it runs faster).*

CHAPTER 2

Entity Relationship Model

2.1 Basic Concepts of E-R Model

2.1.1 The Philosophy of Database Design

Before writing a single line of SQL code, a database architect must mathematically map out exactly how the real-world business operates. The **Entity-Relationship (E-R) Model** is a high-level conceptual data model. It visualizes the entire database architecture as a collection of real-world "Entities" and the mathematical "Relationships" between them.

2.1.2 1. Entity and Entity Sets

- **Entity:** An entity is a "thing" or "object" in the real world that is distinguishable from all other objects. It can be physical (e.g., a specific person, a specific car) or conceptual (e.g., a university course, a bank loan). - **Entity Set:** A collection of entities of the exact same type that share the same properties (e.g., The set of all ‘Students‘ currently enrolled in the university). In a relational database, an Entity Set eventually becomes a **Table**.

2.1.3 2. Attributes

An **Attribute** is a specific property or characteristic that describes an entity. For the ‘Student‘ entity set, the attributes might be ‘StudentID‘, ‘Name‘, ‘Address‘, and ‘GPA‘. In a database, these become the **Columns** of the table. - **Simple vs. Composite:** A simple attribute is indivisible (e.g., ‘Age = 20‘). A composite attribute can be broken down into sub-parts (e.g., ‘Name‘ can be divided into ‘First$_{N}ame$‘and‘$Last_{N}ame$‘). $-**Single-valuedvs.Multi-valued:**Asingle-valuedattributeholdsonevalueforoneentity(e.g.,‘Date_{o}f_{B}irth‘).Amulti-valuedattributecanholdmultiplevaluessimultaneously(e.g.,‘Phone_{N}umber‘,sinceastudentcanhavethreedifferentphone**DerivedAttribute:**Anattributewhosevalueisnotpermanentlystoredinthedatabase,butismathematicallycalculatedfr$

2.1.4 3. Relationship Sets

A **Relationship** is a mathematical association among two or more entities. For example, the student "John" (an entity in the ‘Student‘ set) *is enrolled in* the course "Database Management" (an entity in the ‘Course‘ set). - **Recursive Relationship:** When an entity set has a relationship with itself. (e.g., An ‘Employee‘ table where one employee *supervises* another employee in the exact same table). - **Degree of a Relationship Set:** The number of entity sets participating in a relationship. - Degree 2 (Binary): ‘Student‘ — *Enrolls* — ‘Course‘. (Most common). - Degree 3 (Ternary): ‘Supplier‘ — *Supplies* — ‘Part‘ — *To* — ‘Project‘. - **Participation:** - *Total Participation:* Every entity in the set MUST participate in the relationship. (e.g., Every ‘Loan‘ must belong to at least one ‘Customer‘). - *Partial Participation:* Only some entities participate. (e.g., Not every ‘Employee‘ manages a ‘Department‘).

2.2 Mapping Cardinality & ER Diagrams

2.2.1 Mapping Cardinality

When drawing a relationship between two entity sets (like ‘A‘ and ‘B‘), the **Mapping Cardinality** defines the absolute maximum number of relationship instances that an entity can participate in. There are four possible mathematical constraints:

1. **One-to-One (1:1):** An entity in A is associated with exactly one entity in B, and an entity in B is associated with exactly one entity in A. *(Example: ‘President‘ — rules — ‘Country‘. A country has exactly one president, and that president rules exactly one country).* 2. **One-to-Many (1:M):** An entity in A can be associated with any number of entities in B. But an entity in B can be associated with at most one entity in A. *(Example: ‘Department‘ — employs — ‘Professor‘. A department employs many professors, but each professor is assigned to exactly one department).* 3.

Many-to-One (M:1): The reverse of 1:M. *(Example: ‘Student‘ — takes — ‘Exam‘. Many students take the same exam, but each student’s exam paper belongs to only one student).* 4. **Many-to-Many (M:N):** An entity in A can be associated with many entities in B, and an entity in B can be associated with many entities in A. *(Example: ‘Student‘ — enrolls_in — ‘Course‘. A student takes many courses, and a course contains many students).*

2.2.2 ER Diagrams (The Blueprint Notation)

To standardize database architecture, Peter Chen invented a visual diagramming language in 1976. This is the **ER Diagram**. - **Rectangles:** Represent Entity Sets (e.g., ‘STUDENT‘). - **Ellipses (Ovals):** Represent Attributes (e.g., ‘Name‘). - *Underlined Ellipse:* Primary Key attribute (e.g., ‘Roll$_No$‘). - *Double Ellipse:* Multi-valued attribute (e.g., ‘Phone‘). - *Dashed Ellipse:* Derived attribute (e.g., ‘Age‘). -

CHAPTER 3

Structured Query Language

3.1 Introduction and Basic SQL Commands

3.1.1 Structured Query Language (SQL)

SQL is the standard programming language specifically designed to manage and query relational databases. Unlike Java or Python, which are procedural languages (you tell the computer *how* to do something step-by-step), SQL is a **Declarative Language**. You simply tell the database *what* you want (e.g., "Give me all students with GPA > 3.0"), and the DBMS figures out the fastest way to fetch it.

3.1.2 Categories of SQL Commands

SQL commands are grouped into distinct categories based on their function.

1. **Data Definition Language (DDL):** These commands manipulate the structure (the schema) of the database, not the actual data rows. - ‘CREATE TABLE Student (ID INT, Name VARCHAR(50));‘ (Creates a new, empty table). - ‘ALTER TABLE Student ADD Age INT;‘ (Adds a new column to an existing table). - ‘TRUNCATE TABLE Student;‘ (Deletes all rows incredibly fast, but keeps the empty table structure). - ‘DROP TABLE Student;‘ (Completely deletes the table and all its data from the hard drive).

2. **Data Manipulation Language (DML):** These commands manipulate the actual physical data rows inside the tables. - ‘INSERT INTO Student VALUES (1, ’John’);‘ (Adds one new row). - ‘UPDATE Student SET Name = ’Jane’ WHERE ID = 1;‘ (Modifies existing data). - ‘DELETE FROM Student WHERE ID = 1;‘ (Removes specific rows based on a condition).

3. **Data Query Language (DQL):** This consists of a single, massively powerful command used to retrieve data without changing it. - ‘SELECT Name FROM Student WHERE Age > 18;‘ (Retrieves data).

4. **Data Control Language (DCL / Privilege Commands):** These commands are used by the DBA to manage security and access control. - ‘GRANT SELECT ON Student TO User$_Bob$;‘($GivesBobpermissiontoreadthetable$). – ‘$REVOKESELECTONStudentFROMUser_Bob$;‘($Takesthatpermissionaway$).

5. **Miscellaneous Clauses:** - ‘DESCRIBE Student;‘ (Shows the structure/columns of a table). - ‘SELECT DISTINCT City FROM Student;‘ (Removes duplicate cities from the output). - ‘ORDER BY Age DESC;‘ (Sorts the output from oldest to youngest). - ‘GROUP BY City;‘ (Aggregates data, e.g., finding the total number of students per city).

3.2 SQL Views, Functions, and Operators

3.2.1 SQL Views (Virtual Tables)

A **View** is a virtual table whose contents are generated dynamically by a SQL query. The view itself does not store any physical data on the hard drive; it merely saves the ‘SELECT‘ statement in the database. - *Use Case:* Security and Simplicity. If a table contains sensitive data (like ‘Salary‘), the DBA can create a View that only selects the ‘Name‘ and ‘Department‘ columns. Users can query the View just like a normal table, completely oblivious that the ‘Salary‘ column even exists. - *Syntax:* ‘CREATE VIEW Public$_Employee_View AS SELECT Name, Department FROM Employee$;‘

3.2.2 SQL Functions

Functions are built-in tools that perform calculations on data within a query. They are divided into two types: 1. **Scalar Functions:** Operate on a single row and return a single value for that row. - ‘UPPER(Name)‘ (Converts name to uppercase). - ‘LENGTH(Name)‘ (Returns the number of characters in the name). - ‘ROUND(Price, 2)‘ (Rounds a decimal to 2 places). 2. **Aggregate (Group) Functions:** Operate on a massive column of numbers across

multiple rows, and return a single summary value. - ‘SUM(Salary)‘ (Adds up all salaries). - ‘AVG(Salary)‘ (Calculates the mathematical average). - ‘MAX(Salary)‘ and ‘MIN(Salary)‘. - ‘COUNT(*)‘ (Counts the total number of rows in the table).

3.2.3 SQL Operators

Operators are used in the ‘WHERE‘ clause to filter data mathematically. - **Arithmetic:** ‘+‘, ‘-‘, ‘*‘, ‘/‘. (e.g., ‘WHERE Salary * 1.1 > 50000‘). - **Comparison:** ‘=‘, ‘<‘, ‘>‘, ‘<=‘, ‘>=‘, ‘<>‘ (Not equal). - **Logical:** ‘AND‘, ‘OR‘, ‘NOT‘. - **Special Operators:** - ‘BETWEEN 10 AND 20‘ (Selects values within a range, inclusive). - ‘IN (’NY’, ’CA’, ’TX’)‘ (Matches any value in the specific list). - ‘LIKE ’A

3.3 Set Operators

SQL **Set Operators** are used to mathematically combine the results of two completely separate ‘SELECT‘ queries into a single, unified output table.

Critical Rule: To use a set operator, the two queries must be "Union Compatible." This means they must select the exact same number of columns, and those columns must have matching data types in the same order.

Consider Query A (Selects Math students) and Query B (Selects Physics students).

1. **UNION:** Combines the results of A and B, and **automatically removes all duplicate rows**. If John takes both Math and Physics, he will only appear once in the final output. *(Syntax: ‘SELECT Name FROM Math UNION SELECT Name FROM Physics;‘)*

2. **UNION ALL:** Combines the results of A and B, but **keeps all duplicates**. If John takes both subjects, his name will appear twice in the output. This is significantly faster for the computer to execute than a standard UNION because the database engine does not have to spend CPU time searching for and deleting duplicates.

3. **INTERSECT:** Returns only the rows that are present in BOTH Query A and Query B. It effectively finds the overlap. In our example, it will output only the students who are taking *both* Math and Physics simultaneously.

4. **MINUS (or EXCEPT in some databases):** Returns the rows from Query A that are NOT present in Query B. It subtracts the second set from the first. *(Syntax: ‘SELECT Name FROM Math MINUS SELECT Name FROM Physics;‘)* This will output students who are taking Math, but mathematically excludes anyone who is also taking Physics.

3.4 Joins

The most powerful feature of a Relational Database is the ability to connect data scattered across multiple tables. A **JOIN** operation horizontally stitches rows from two different tables together based on a related column (usually a Primary Key to Foreign Key relationship).

Assume we have a ‘STUDENT‘ table (RollNo, Name, DeptID) and a ‘DEPARTMENT‘ table (DeptID, DeptName).

1. **INNER JOIN (The Most Common):** This returns only the rows where there is a perfect match in both tables. If a student has a ‘DeptID‘ of 5, but Department 5 does not exist in the Department table, that student is completely excluded from the output. *(Syntax: ‘SELECT Name, DeptName FROM Student INNER JOIN Department ON Student.DeptID = Department.DeptID;‘)*

2. **LEFT OUTER JOIN:** This returns ALL rows from the Left table (‘STUDENT‘), and only the matched rows from the Right table (‘DEPARTMENT‘). If a student has not been assigned a department yet, they will still appear in the final output, but their ‘DeptName‘ column will show as ‘NULL‘ (empty).

3. **RIGHT OUTER JOIN:** The exact inverse of the Left Join. It returns ALL rows from the Right table (‘DEPARTMENT‘), even if no students are enrolled in that department. The student names for that department will show as ‘NULL‘.

4. **FULL OUTER JOIN:** Returns all rows from both tables. It combines the results of both Left and Right joins.

5. **CROSS JOIN (Cartesian Product):** This joins every single row in Table A with every single row in Table B. If Table A has 100 rows and Table B has 100 rows, the output will be a massive 10,000 rows. It is rarely used in business applications because it creates an exponential explosion of mostly meaningless data.

3.5 SQL Constraints

3.5.1 The Need for Constraints

If users are allowed to type whatever they want into a database, the data will quickly become corrupted garbage (e.g., typing "-50" for an employee’s age, or spelling "New York" three different ways). **Constraints** are strict mathematical

rules enforced by the DBMS at the moment data is inserted or updated. If the data violates the constraint, the DBMS instantly rejects the transaction and throws an error.

3.5.2 1. Domain Integrity Constraints

These rules govern the validity of data inside a single, specific column. - **NOT NULL:** Prevents a column from being left blank. (e.g., A student's 'Name' cannot be NULL). - **CHECK:** Applies a specific mathematical or logical condition. (e.g., 'CHECK (Age >= 18)' ensures no minors are entered into the employee table).

3.5.3 2. Entity Integrity Constraints

These rules ensure that every single row in a table is uniquely identifiable. - **UNIQUE:** Guarantees that all values in a column are completely distinct. No two rows can have the same value. (e.g., Two employees cannot share the same 'Email$_{A}ddress$'). $-**PRIMARYKEY:**Theultimateidentifier.Itisamathematicalcombinationof`NOTNULL`and`UNIQUE`.Everytablemusthaveone,andonl$

3.5.4 3. Referential Integrity Constraints (Foreign Keys)

This is the most critical constraint in relational databases. It governs the relationships *between* two different tables. - **FOREIGN KEY:** A column in one table that points directly to the Primary Key of another table. - *The Rule:* A Foreign Key value MUST either perfectly match an existing Primary Key in the parent table, or it must be NULL. - *Example:* If the 'STUDENT' table has a Foreign Key 'Dept$_{I}D$`pointingtothe`$DEPARTMENT$`table`$,auserisphysicallypreventedfrominsertingastudentwith`Dept$_{I}D=$99`if`$Department99doesnotactuallyexistintheDepartmenttable.Thisprevents"orphanrecords"andmaintainsperfectrela$

CHAPTER 4

Normalization

4.1 Anomalies Created by Poor Database Design

4.1.1 The Problem with Giant Tables

When novice developers design a database, they often make the catastrophic mistake of putting all their data into one massive, unorganized table (similar to a giant Excel spreadsheet). For example, a single table storing ‘StudentID‘, ‘StudentName‘, ‘CourseID‘, ‘CourseName‘, and ‘ProfessorName‘. This "flat-file" design creates severe data redundancy (repetition), which leads directly to three types of catastrophic errors known as **Data Anomalies**.

1. **Insertion Anomaly:** This occurs when you are physically prevented from adding data to the database because other unrelated data is missing. *Example:* If the giant table uses a combined Primary Key of (‘StudentID‘ + ‘CourseID‘), you cannot add a brand new course to the catalog until at least one student enrolls in it, because the ‘StudentID‘ part of the Primary Key cannot be NULL.

2. **Update Anomaly:** This occurs when updating a single piece of real-world information requires the system to update hundreds of thousands of identical rows. *Example:* Professor Smith teaches 500 students. His name "Smith" is copied 500 times in the giant table. If he changes his name to "Johnson", the DBA must write a query to update all 500 rows. If the system crashes at row 250, half the students are taught by Smith and half by Johnson. The database is now mathematically inconsistent and corrupt.

3. **Deletion Anomaly:** This occurs when deleting one piece of data accidentally triggers the deletion of entirely unrelated, critical data. *Example:* A student is the only person enrolled in a highly specialized Advanced Physics course. If that student drops out and their row is deleted, the entire record of the Advanced Physics course (its name, ID, and Professor) is permanently erased from the university database.

4.2 Normalization: Definition and Goals

4.2.1 What is Normalization?

Normalization is the systematic, mathematical process of breaking down a massive, poorly designed table into multiple smaller, highly efficient tables, and then linking them back together using Foreign Keys. It was invented by E.F. Codd (the creator of the relational model) to impose strict logic on data structures.

4.2.2 The Goals of Normalization

The primary objective of normalization is to completely eliminate the Data Anomalies (Insertion, Update, Deletion) discussed previously.

Specifically, normalization aims to achieve: 1. **Elimination of Data Redundancy:** Storing every single fact in exactly one place. If a professor’s name needs to be changed, it should only take exactly one SQL update on exactly one row. 2. **Minimization of Null Values:** Poorly designed tables are often riddled with empty (NULL) cells, which waste hard drive space and confuse aggregate SQL functions (like ‘AVG()‘). Breaking tables down ensures data is dense. 3. **Prevention of Unwanted Dependencies:** Ensuring that every non-key column in a table depends *only* on the Primary Key of that table, and nothing else. 4. **Optimization for DML Operations:** Because the tables are smaller and lack redundancy, ‘INSERT‘, ‘UPDATE‘, and ‘DELETE‘ queries execute incredibly fast.

4.2.3 The Trade-off (Denormalization)

While normalization speeds up Data Manipulation Language (DML) operations (updates/deletes), it actually *slows down* Data Query Language (DQL) operations (selects). If data is split across 5 normalized tables, retrieving the full picture requires the CPU to execute 5 complex SQL ‘JOIN‘ operations. Therefore, in massive Data Warehouses used exclusively for reading data, engineers will deliberately "Denormalize" the database to speed up read performance.

4.3 Functional Dependencies

To perform normalization, we must first mathematically define how columns relate to each other. We do this using **Functional Dependencies (FD)**.

4.3.1 Prime vs. Non-Prime Attributes

- **Prime Attribute:** Any column that is part of a Candidate Key or Primary Key. - **Non-Prime Attribute:** A regular data column that is not part of any key (e.g., ‘Address‘, ‘Phone‘).

4.3.2 Functional Dependency (X → Y)

A functional dependency exists when the value of one column (X) uniquely and mathematically determines the exact value of another column (Y). We write this as ‘X → Y‘($XdeterminesY$).$*Example:*‘Social_{S}ecurity_{N}umber \rightarrow Name‘.Ifyouknowthe SSN, thereisonlyonepossibleNameitcouldbe.Therefore, Nameisfunctionallydependentonon SSN.$

4.3.3 Types of Functional Dependencies

1. **Fully Functional Dependency:** This applies when a table has a *Composite Primary Key* (a key made of two columns, like ‘StudentID‘ + ‘CourseID‘). An attribute is fully functionally dependent if it depends on the ENTIRE composite key. *Example:* ‘(StudentID, CourseID) → $Grade‘.Youcannotfindthegradewithjustthe StudentID(theytakemanycourses), andyoucannotfinditwithjustthe Course$ $both*$.

2. **Partial Functional Dependency:** This is a severe design flaw. It happens when a non-prime attribute depends on *only a portion* of the composite primary key. *Example:* If the primary key is ‘(StudentID, CourseID)‘, and the table includes a column ‘$Student_{P}hone‘.The‘Student_{P}hone‘dependsentirelyon‘StudentID‘; ithasabsolutelynothingtodowiththe‘CourseID‘.This$

3. **Transitive Dependency:** This is another severe flaw. It happens when a non-prime attribute depends on *another non-prime attribute*, rather than depending directly on the Primary Key. *Example:* ‘StudentID → $Department_{C}ode \rightarrow Department_{B}uilding‘.Here, theBuildingdependsontheDeptCode, nottheStudent.Thisisatransitive$

4.4 Normal Forms (1NF, 2NF, 3NF, BCNF)

Normalization proceeds in sequential stages called Normal Forms. You cannot reach a higher form without first satisfying all the lower forms.

4.4.1 First Normal Form (1NF)

Rule: A table is in 1NF if and only if every cell contains an **Atomic (Indivisible) Value**. - A relational database cannot store lists, arrays, or comma-separated values in a single cell. - *Violation:* A ‘$Phone_{N}umbers‘columncontaining$”555 − 1234, 555 − 9876”. − $*Solution:*Createmultiplerowsforthatstudent, oneforeachphonenumber, orcreateaseparate‘PHONE‘table.$

4.4.2 Second Normal Form (2NF)

Rule: A table is in 2NF if it is already in 1NF, AND it contains **No Partial Dependencies**. - Every non-prime attribute must depend on the *entire* composite primary key. (Note: If a table has a single-column primary key, it is automatically in 2NF). - *Violation:* Table ‘(StudentID, CourseID, Grade, StudentName)‘. ‘StudentName‘ only depends on ‘StudentID‘. - *Solution:* Split into two tables: ‘ENROLLMENT (StudentID, CourseID, Grade)‘ and ‘STUDENT (StudentID, StudentName)‘.

4.4.3 Third Normal Form (3NF)

Rule: A table is in 3NF if it is in 2NF, AND it contains **No Transitive Dependencies**. - Every non-prime attribute must depend *only* on the Primary Key, the whole Primary Key, and nothing but the Primary Key. Non-prime attributes cannot determine other non-prime attributes. - *Violation:* ‘EMPLOYEE (EmpID, Name, DeptID, DeptName)‘. ‘DeptName‘ depends on ‘DeptID‘, which is not the primary key. - *Solution:* Split into ‘EMPLOYEE (EmpID, Name, DeptID)‘ and ‘DEPARTMENT (DeptID, DeptName)‘.

4.4.4 Boyce-Codd Normal Form (BCNF)

Rule: BCNF is a stricter version of 3NF. A table is in BCNF if for every functional dependency 'X $\rightarrow Y$', $Xisa**Superkey**(acandidatekey).-Inrareedgecases(usuallyinvolvingoverlappingcompositecandidatekeys), atablecanbein3Iprimeattribute.BCNFforbidsthis.$

CHAPTER 5

Transaction Management

5.1 Basic Transaction Concepts

5.1.1 What is a Transaction?

In a database, a **Transaction** is a logical unit of work that contains one or more SQL statements (usually DML like ‘UPDATE‘ or ‘INSERT‘). *Example:* Transferring $100 from Account A to Account B is a single transaction, but it requires two physical SQL operations: 1. ‘UPDATE Account SET Balance = Balance - 100 WHERE ID = A;‘ 2. ‘UPDATE Account SET Balance = Balance + 100 WHERE ID = B;‘

If the database server crashes exactly between step 1 and step 2, Account A loses $100, but Account B never receives it. The money vanishes. Transaction Management exists exclusively to prevent this catastrophic scenario.

5.1.2 The ACID Properties

To ensure perfect data integrity even during hardware failures, a DBMS guarantees that every transaction strictly follows the four **ACID** properties:

1. **Atomicity (The "All or Nothing" Rule):** A transaction is treated as a single, indivisible atomic unit. Either *all* of its SQL operations execute successfully and are saved to the hard drive, or *none* of them are. If a crash happens halfway through, the DBMS automatically reverses (Rolls Back) the partial changes upon reboot. 2. **Consistency:** A transaction must take the database from one valid, mathematically consistent state to another valid state. It cannot break any constraints (like Foreign Keys). 3. **Isolation:** When multiple users execute transactions simultaneously, the system must ensure that they do not interfere with each other. Each transaction must feel like it is the only one running on the server. 4. **Durability:** Once a transaction completes successfully (it is ‘COMMITTED‘), the changes are permanently written to non-volatile storage (the hard drive). Even if someone unplugs the server 1 millisecond later, the data is safe forever.

5.1.3 Transaction State Transition Diagram

During execution, a transaction passes through several states: - **Active:** The initial state; executing SQL commands. - **Partially Committed:** The final SQL command was executed, but changes are still in RAM, not yet written to the hard drive. - **Committed:** Changes are permanently saved to the hard drive (Durability achieved). - **Failed:** A hardware crash or constraint violation occurred during the Active state. - **Aborted:** The DBMS has successfully rolled back the failed changes, restoring the database to its previous safe state.

5.1.4 The Transaction Log

How does the DBMS achieve Atomicity and undo a failed transaction? It uses a **Transaction Log**. This is a highly secure, append-only file on the hard drive. Before the DBMS modifies any actual table data, it writes a permanent record to the Log detailing exactly what it is *about* to do (e.g., "Change Account A from 500 to 400"). If a crash occurs, the recovery manager reads the Log backwards to undo any uncommitted changes.

5.2 Concurrency Control

5.2.1 The Need for Concurrency

In a modern enterprise, thousands of users access the database simultaneously (e.g., thousands of people booking flights on an airline website). If the DBMS only allowed one transaction to run at a time (serial execution), the system would be unbearably slow. The DBMS must allow **Concurrent Execution** (overlapping transactions) to maximize CPU utilization and throughput. However, unmanaged concurrency causes severe data corruption.

5.2.2 Problems of Concurrency Control

If multiple transactions read and write the exact same data simultaneously without strict management, three classic errors occur:

1. **The Lost Update Problem:** Two users read the exact same data, modify it, and write it back. The second user's write overwrites the first user's write, effectively erasing the first transaction entirely. *(Example: Ticket agent A and Agent B both see 1 seat left on a flight. They both sell the seat and update the database to 0. Two people show up at the airport for the same seat).*

2. **The Dirty Read Problem (Uncommitted Dependency):** Transaction T_1 updates a value but has not yet committed it. Transaction T_2 reads that updated value and uses it in a calculation. Then, T_1 crashes and is Rolled Back. T_2 is now relying on a "dirty" value that mathematically never existed.

3. **The Unrepeatable Read Problem:** Transaction T_1 reads a value (e.g., Balance = 100). Before T_1 finishes, Transaction T_2 modifies that value and commits (Balance = 50). Later, T_1 reads the exact same value again to double-check, but gets a different number. This breaks the Isolation property.

5.2.3 Schedules

A **Schedule** is the exact chronological execution sequence of operations (‘READ‘, ‘WRITE‘, ‘COMMIT‘) from multiple concurrent transactions. The **Concurrency Control Manager** is the subsystem inside the DBMS responsible for generating safe schedules that allow concurrency without causing any of the three problems listed above.

5.3 Serializability of Transactions

5.3.1 Serial vs. Concurrent Schedules

- **Serial Schedule:** Transactions execute completely one after the other. T_1 finishes entirely before T_2 begins. A serial schedule is mathematically guaranteed to be 100% safe and correct (no Lost Updates or Dirty Reads). However, it offers terrible performance. - **Concurrent Schedule:** The operations of T_1 and T_2 are interleaved to maximize CPU efficiency.

5.3.2 The Concept of Serializability

How does the DBMS know if a highly interleaved concurrent schedule is actually safe to run? It uses the mathematical concept of **Serializability**. A concurrent schedule is considered "Serializable" if its final mathematical outcome is 100% identical to the outcome of *some* Serial schedule. If the interleaved execution produces the exact same database state as running them one by one, then the interleaved execution is proven to be safe.

5.3.3 Conflict Serializability

To determine if a schedule is serializable, the DBMS analyzes "Conflicting Operations." Two operations conflict if they belong to different transactions, access the exact same data item (e.g., Variable X), and at least one of them is a ‘WRITE‘ operation. - ‘READ(X)‘ and ‘READ(X)‘: No conflict. - ‘READ(X)‘ and ‘WRITE(X)‘: Conflict (The order matters). - ‘WRITE(X)‘ and ‘WRITE(X)‘: Conflict (The order matters).

If a concurrent schedule can be mathematically transformed into a Serial Schedule simply by swapping non-conflicting operations, the schedule is **Conflict Serializable**. The DBMS will confidently execute it knowing it is safe. If it cannot be transformed, the schedule is unsafe, and the Concurrency Control Manager will pause or abort one of the transactions to prevent data corruption.

5.4 Locking Methods for Concurrency Control

To guarantee that generated schedules are serializable and free from Lost Updates and Dirty Reads, the DBMS physically prevents transactions from interfering with each other using **Locks**.

5.4.1 The Concept of Locking

A lock is a variable associated with a data item (like a row in a table) that dictates whether other transactions are allowed to access it. Before a transaction can read or write data, it must formally request a lock from the Lock Manager.

5.4.2 Types of Locks

To allow as much concurrency as safely possible, most systems use two distinct types of locks: 1. **Shared Lock (Read Lock - 'S'):** If transaction T_1 only wants to 'READ' a row, it requests a Shared Lock. If granted, T_1 can read the data. Crucially, if T_2 also wants to 'READ' the same row, it can also acquire a Shared Lock. Many transactions can hold a Shared Lock simultaneously because reading does not corrupt data. 2. **Exclusive Lock (Write Lock - 'X'):** If transaction T_1 wants to 'UPDATE' or 'WRITE' a row, it MUST request an Exclusive Lock. If granted, T_1 is the *only* transaction allowed to touch that row. No other transaction can read it, and no other transaction can write to it, until T_1 releases the lock. This instantly prevents the Lost Update problem.

5.4.3 Two-Phase Locking Protocol (2PL)

Simply using locks is not enough; transactions must follow a strict mathematical protocol for acquiring and releasing them to guarantee serializability. The industry standard is the **Two-Phase Locking (2PL)** protocol.

Under 2PL, a transaction executes in two distinct phases: 1. **Growing Phase:** The transaction is allowed to request and acquire new locks as it needs them, but it is strictly forbidden from releasing any locks. 2. **Shrinking Phase:** Once the transaction releases its very first lock, it enters the shrinking phase. It can continue to release locks, but it is strictly forbidden from ever acquiring a new lock.

Mathematical Guarantee: If every transaction in a system strictly obeys the 2PL protocol, the resulting concurrent schedule is mathematically guaranteed to be Conflict Serializable (perfectly safe).

5.4.4 The Danger of Deadlock

While 2PL guarantees safety, it introduces a new critical failure called a **Deadlock**. - T_1 holds an Exclusive Lock on Row A, and wants a lock on Row B. - T_2 holds an Exclusive Lock on Row B, and wants a lock on Row A. Both transactions pause indefinitely, waiting for the other to release the lock. The DBMS must use a Deadlock Detection algorithm (analyzing a Wait-For Graph) to detect this infinite freeze, forcefully abort one of the transactions, and roll it back to break the deadlock.