Dbms (1333204) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Introduction to Database Systems

In this chapter, we focus on the foundational methods used to design and analyze database systems. This includes transitioning from traditional file processing to database approaches, classifying system architectures, designing basic Entity-Relationship models, and identifying keys based on given requirements.

1.1 Database Approach vs File Processing

Methodology:

Identifying the Storage Approach To determine whether a scenario requires a traditional file processing system or a Database Management System (DBMS), use the following steps:

1. **Analyze Data Redundancy**: If data is duplicated across multiple files without synchronization, it resembles a file system. If centralized and normalized, it resembles a DBMS.
2. **Analyze Data Isolation**: If data is scattered in various formats, making it hard to retrieve across boundaries, it indicates a file system.
3. **Examine Concurrent Access**: If multiple users need to safely update the data simultaneously, a DBMS with concurrency control is required.
4. **Check Integrity Constraints**: If complex constraints and relationships must be automatically enforced by the system, a DBMS is necessary.

Worked Example:

Scenario Analysis for Storage Choice A small library currently uses spreadsheets to store books and student details. Whenever a student borrows a book, their name is typed again into the 'borrow' spreadsheet. When a student changes their phone number, it must be updated in both the 'students' and 'borrow' spreadsheets. Analyze the issues and determine the appropriate system.
Solution:

1. **Analyze Data Redundancy**: The student's name and details are duplicated in both the 'students' and 'borrow' spreadsheets. This violates the principle of single-source truth.
2. **Analyze Data Isolation**: Data is isolated in separate spreadsheet files, meaning a combined query (e.g. list of all books borrowed by a specific major) requires manual cross-referencing.
3. **Check Integrity Constraints**: If a student is deleted from the 'students' spreadsheet, there is no automatic check to prevent their records from remaining in the 'borrow' spreadsheet.
4. **Conclusion**: The library is using a File Processing approach which suffers from data redundancy and lack of integrity constraints. A Database Management System should be implemented to centralize data and enforce referential integrity.

1.2 Database System Architecture

Methodology:

Categorizing Database Architectures To classify a database architecture into 1-Tier, 2-Tier or 3-Tier configurations, apply these rules:

1. **1-Tier Architecture**: The database and the application reside on the exact same local machine. The user accesses the database directly.
2. **2-Tier Architecture**: The application runs on a client machine and communicates directly with a database running on a separate server.
3. **3-Tier Architecture**: The client application interacts with an intermediate application server (middle tier), which in turn processes business logic and communicates with the database server.

Worked Example:

Architecture Classification Given the following three deployment scenarios, classify each into the correct database architecture tier. Scenario A: A desktop point-of-sale software installed on a shop computer that reads and writes to an Access database file stored on the same hard drive. Scenario B: A mobile banking app that sends REST API requests to a cloud-hosted web server which then queries a database to retrieve the balance. Scenario C: A Java desktop application installed on employee laptops that establishes a JDBC connection directly to a central database server in the corporate network.
Solution:

1. **Scenario A**: The application and database are on the exact same machine. Thus, it is a **1-Tier Architecture**.
2. **Scenario B**: The mobile app (client) talks to a web server (application server), which talks to the database (database server). Thus, it is a **3-Tier Architecture**.
3. **Scenario C**: The Java application (client) communicates directly with the database server via JDBC without an intermediary business logic server. Thus, it is a **2-Tier Architecture**.

1.3 Entity-Relationship Model Components

Methodology:

Identifying ER Model Components When reading a system requirement, identify the components of an Entity-Relationship (ER) model as follows:

1. **Entities**: Identify nouns representing independent objects or concepts (e.g. Student, Course).
2. **Attributes**: Identify properties or characteristics that describe an entity (e.g. Name, Age). Note if they are Key (unique), Multivalued (multiple entries), or Derived (calculated).
3. **Relationships**: Identify verbs or actions that connect two or more entities (e.g. Enrolls, Teaches).
4. **Cardinality**: Determine the numerical relationship between entities (1:1, 1:N, or M:N).

Worked Example:

Component Extraction from Requirements Extract the Entities, Attributes, Relationships and Cardinality from the following requirement: "A university wants to track Instructors and Departments. An Instructor has an ID, a Name and a List of Phone Numbers. A Department has a unique DeptCode and a Location. Each Instructor belongs to exactly one Department, but a Department can have many Instructors working

in it."
Solution:

1. **Entities**:
   - Instructor
   - Department
2. **Attributes**:
   - For Instructor: ID (Key), Name (Simple), List of Phone Numbers (Multivalued).
   - For Department: DeptCode (Key), Location (Simple).
3. **Relationships**: "Belongs to" (or "Works in"), which connects Instructor and Department.
4. **Cardinality**: The requirement states "Instructor belongs to exactly one Department" (1) and "Department can have many Instructors" (N). Thus, the relationship is Many-to-One (M:1) from Instructor to Department, or One-to-Many (1:N) from Department to Instructor.

1.4 Constructing Entity-Relationship Diagrams

Methodology:

ER Diagram Construction Rules Follow these mapping rules to convert identified components into a visual ER Diagram representation:

1. **Entity Sets** are represented by **Rectangles**.
2. **Attributes** are represented by **Ovals** connected to their entity.
3. **Primary Key Attributes** are represented by **Ovals with underlined text**.
4. **Multivalued Attributes** are represented by **Double Ovals**.
5. **Derived Attributes** are represented by **Dashed Ovals**.
6. **Relationships** are represented by **Diamonds** connecting the participating entities.

Worked Example:

Basic ER Diagram Design Design the logical structure of an ER diagram for a system where 'Customers' place 'Orders'. A Customer has a unique CustID, Name and a derived Age. An Order has an OrderID and an Amount. A Customer can place multiple Orders, but each Order belongs to only one Customer.
Solution:

1. **Identify Components**:
   - Entities: Customer, Order
   - Customer Attributes: CustID (Key), Name (Simple), Age (Derived)
   - Order Attributes: OrderID (Key), Amount (Simple)
   - Relationship: "Places"
   - Cardinality: 1:N from Customer to Order.
2. **Apply Visual Rules (Description of visual graph)**:
   - Draw a rectangle for **Customer** and a rectangle for **Order**.
   - Connect a diamond named **Places** between Customer and Order.

- Draw an oval for **CustID** (underline the text) and connect to Customer.
- Draw an oval for **Name** and connect to Customer.
- Draw a dashed oval for **Age** and connect to Customer.
- Draw an oval for **OrderID** (underline the text) and connect to Order.
- Draw an oval for **Amount** and connect to Order.
- Draw a line from Customer to Places with a '1' and from Places to Order with an 'N'.

1.5 Database Keys Identification

Methodology:

Key Identification Strategy To identify keys from a set of attributes in a table, follow these steps:

1. **Super Key**: Identify any combination of attributes that uniquely identifies a row.
2. **Candidate Key**: From the set of Super Keys, identify those that are minimal (no subset of the attributes can uniquely identify the row). There can be multiple Candidate Keys.
3. **Primary Key**: Choose exactly one Candidate Key to serve as the main identifier for the table.
4. **Alternate Key**: All Candidate Keys that were not selected as the Primary Key become Alternate Keys.
5. **Foreign Key**: Identify attributes whose values are required to match the Primary Key of another table, establishing a relationship.

Worked Example:

Finding Keys from Data Attributes Consider a table `Employee` with the following attributes: `EmpID`, `SSN`, `Email`, `Name`, `DepartmentID`. Assume `EmpID`, `SSN`, and `Email` are strictly unique for every employee. `DepartmentID` matches a department in the `Department` table. Identify the Candidate Keys, Primary Key, Alternate Keys, and Foreign Keys.
Solution:

1. **Identify Super Keys**: Any combination containing `EmpID`, `SSN`, or `Email` (e.g. {`EmpID`}, {`EmpID`, `Name`}, {`SSN`, `Email`}).
2. **Identify Candidate Keys**: The minimal unique identifiers are {`EmpID`}, {`SSN`}, and {`Email`}.
3. **Select Primary Key**: We can choose {`EmpID`} as the most natural Primary Key since it is specifically designed to identify the entity within the system.
4. **Identify Alternate Keys**: The remaining candidate keys, {`SSN`} and {`Email`}, become the Alternate Keys.
5. **Identify Foreign Keys**: {`DepartmentID`} refers to another table (`Department`), so it acts as a Foreign Key.

CHAPTER 2

ER Model and Relational Algebra

2.1 Entity-Relationship (ER) Modeling

The Entity-Relationship (ER) model is a high-level conceptual data model used to define the data elements and relationships for a specified system. In problem-solving, constructing an ER diagram from a set of requirements is a fundamental computational task.

Methodology:

Steps to Construct an ER Diagram

1. **Identify Entity Sets:** Read the problem statement and identify the main objects or concepts (e.g. Student, Course, Department).
2. **Identify Attributes:** Determine the properties of each entity. Identify the primary key (unique identifier) for each entity set.
3. **Identify Relationship Sets:** Find the associations between entity sets (e.g. Student "enrolls in" Course).
4. **Determine Cardinality Ratios:** For each relationship determine how many instances of one entity can associate with instances of another entity (1:1, 1:N, N:1, or M:N).
5. **Identify Participation Constraints:** Determine if the participation of an entity in a relationship is total (mandatory) or partial (optional).

Worked Example:

Constructing an ER Diagram for a Hospital **Problem Statement:** A hospital requires a database to manage its operations. 1. The hospital has several doctors. Each doctor has a unique DoctorID Name and Specialization. 2. The hospital registers multiple patients. Each patient has a PatientID Name Age and Address. 3. Doctors treat patients. A doctor can treat multiple patients and a patient can be treated by multiple doctors. 4. Each treatment has a specific Date and Diagnosis associated with it.
Identify the entities attributes relationships and cardinalities.
Solution:

1. **Entity Sets:**
   - **Doctor**
   - **Patient**
2. **Attributes:**
   - Doctor: DoctorID (Primary Key), Name, Specialization
   - Patient: PatientID (Primary Key), Name, Age, Address
3. **Relationship Set:**

- **Treats**: Associates Doctor and Patient.
- Attributes of relationship 'Treats': Date, Diagnosis (These are descriptive attributes of the M:N relationship).

4. **Cardinality Ratio:**
   - Since a doctor treats multiple patients and a patient is treated by multiple doctors, the cardinality for the **Treats** relationship is **M:N** (Many-to-Many).

2.2 Converting ER Model to Relational Schema

Once the ER diagram is finalized, the next step in database design is to map it to a relational schema (a set of tables).

Methodology:

Mapping ER Model to Relational Schema

1. **Strong Entity Sets:** Create a table for each strong entity. The primary key of the entity becomes the primary key of the table.
2. **Weak Entity Sets:** Create a table with the attributes of the weak entity. The primary key is a composite of the weak entity's partial key and the primary key of its identifying strong entity.
3. **1:1 Relationships:** Add the primary key of one entity as a foreign key in the table of the other entity. (Preferably put the foreign key on the side with total participation).
4. **1:N Relationships:** Add the primary key of the "1" side entity as a foreign key in the table of the "N" side entity.
5. **M:N Relationships:** Create a *new table* for the relationship. The primary key of this new table is a composite key consisting of the primary keys of the participating entities. Include any descriptive attributes of the relationship in this new table.
6. **Multivalued Attributes:** Create a *new table* containing the primary key of the entity and the multivalued attribute itself. The primary key of this new table is the combination of both columns.

Worked Example:

Mapping Hospital ER to Relational Tables **Problem Statement:** Map the Hospital ER model from the previous example into a relational schema.
Entities and Relationship:

- Doctor(DoctorID, Name, Specialization)
- Patient(PatientID, Name, Age, Address)
- Treats(Date, Diagnosis) - M:N relationship between Doctor and Patient.

Solution:

1. **Map Strong Entities:**
   - `Doctor(DoctorID, Name, Specialization)`
   - `Patient(PatientID, Name, Age, Address)`
2. **Map M:N Relationship (Treats):**
   - Create a new table named `Treats`.

- Include primary keys of Doctor and Patient as foreign keys.
- The composite primary key will be `(DoctorID, PatientID)`.
- Add the relationship attributes `Date` and `Diagnosis`.
- `Treats(DoctorID, PatientID, Date, Diagnosis)`

Final Relational Schema:

Relational Tables
Doctor(DoctorID, Name, Specialization)
Patient(PatientID, Name, Age, Address)
Treats(DoctorID, PatientID, Date, Diagnosis)

2.3 Relational Algebra

Relational algebra is a procedural query language that takes instances of relations as input and yields instances of relations as output. It uses mathematical operators to perform queries.

Methodology:

Basic Relational Algebra Operations

1. **Selection (σ):** Selects a subset of rows (tuples) from a relation that satisfy a specific selection condition.
   - Notation: $\sigma_{condition}(R)$
2. **Projection (π):** Selects a subset of columns (attributes) from a relation and eliminates duplicates.
   - Notation: $\pi_{A_1 A_2 \ldots A_n}(R)$
3. **Rename (ρ):** Renames the output relation or its attributes.
   - Notation: $\rho_S(R)$ or $\rho_{S(A_1 \ldots A_n)}(R)$
4. **Cross Product ($\times$):** Also known as Cartesian Product. Combines every tuple of one relation with every tuple of another relation.
   - Notation: $R \times S$

Worked Example:

Writing Basic Relational Algebra Queries **Problem Statement:** Consider the following relation:

Student(EnrollmentNo, Name, Branch, Semester, CPI)

Write relational algebra expressions for the following queries:

1. Find all details of students in the 'Computer' branch.
2. Find the names and CPI of all students.
3. Find the names of students in the 'IT' branch who have a CPI greater than 8.0.

Solution:

1. **Select students in 'Computer' branch:** We need to filter rows, so we use the Selection operator.
   $\sigma_{Branch='Computer'}(\text{Student})$
2. **Find names and CPI:** We need to select specific columns, so we use the Projection operator.
   $\pi_{Name,CPI}(\text{Student})$

3. **Find names of 'IT' branch students with CPI > 8.0:** First filter the rows using Selection with an AND condition ($\wedge$), then extract the Name column using Projection.

   $\pi_{Name}(\sigma_{Branch='IT' \wedge CPI > 8.0}(\text{Student}))$

Methodology:

Advanced Relational Algebra Operations

1. **Set Operations:** Used to combine results of two relations. The relations must be union-compatible (same number of attributes and identical domains).
   - **Union ($\cup$):** Includes all tuples that are in R, in S, or in both.
   - **Intersection ($\cap$):** Includes all tuples that are in both R and S.
   - **Set Difference ($-$):** Includes tuples that are in R but not in S.
2. **Natural Join ($\bowtie$):** Combines two relations over their common attributes. It automatically equates the common attributes and removes one duplicate column.
   - Notation: $R \bowtie S$
3. **Theta Join ($\bowtie_\theta$):** A join operation that combines tuples from two relations based on a specified condition θ (e.g. $<, >, \leq, \geq$).
   - Notation: $R \bowtie_\theta S$

Worked Example:

Queries using Joins and Set Operations **Problem Statement:** Consider the following relational schema for a banking system:

Customer(CustomerID, CustomerName, City)
Account(AccountNo, Balance, CustomerID)
Loan(LoanNo, Amount, CustomerID)

Write relational algebra expressions for the following queries:

1. Find the names of all customers who have an account.
2. Find the names of all customers who have an account in the bank and live in 'Ahmedabad'.
3. Find the Customer IDs of customers who have a loan, an account, or both.
4. Find the names of customers who have a loan but do NOT have an account.

Solution:

1. **Names of customers with an account:** We need to join the Customer and Account tables to find matching records, then project the name. Natural join operates on the common attribute `CustomerID`.

   $\pi_{CustomerName}(\text{Customer} \bowtie \text{Account})$

2. **Customers with an account living in 'Ahmedabad':** Apply selection on the Customer table, then join with Account, and finally project the name.

   $\pi_{CustomerName}(\sigma_{City='Ahmedabad'}(\text{Customer}) \bowtie \text{Account})$

3. **Customer IDs with a loan, an account, or both:** This is a union operation combining Customer IDs from both tables.

   $\pi_{CustomerID}(\text{Account}) \cup \pi_{CustomerID}(\text{Loan})$

4. **Names of customers with a loan but NO account:** First find the Customer IDs with a loan but no account using Set Difference. Then join this result with the Customer table to get the names.

   Let $L\_only = \pi_{CustomerID}(\text{Loan}) - \pi_{CustomerID}(\text{Account})$

   Result: $\pi_{CustomerName}(\text{Customer} \bowtie L\_only)$

CHAPTER 3

Structured Query Language (SQL)

3.1 Data Definition Language (DDL) Commands

Methodology:

Creating Modifying and Deleting Tables SQL provides Data Definition Language (DDL) commands to define and manage the database structure.

1. **CREATE TABLE:** Used to create a new table with specified columns and data types.

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    ...
);
```

2. **ALTER TABLE:** Used to add, modify, or drop columns in an existing table.

```
ALTER TABLE table_name ADD column_name datatype;
ALTER TABLE table_name DROP COLUMN column_name;
ALTER TABLE table_name MODIFY column_name new_datatype;
```

3. **DROP TABLE:** Used to delete an existing table and all its data permanently.

```
DROP TABLE table_name;
```

4. **TRUNCATE TABLE:** Used to remove all records from a table while preserving its structure.

```
TRUNCATE TABLE table_name;
```

Worked Example:

Creating and Modifying an Employee Table Create a table named `Employee` with columns `EmpID` (Primary Key), `Name` (Variable character up to 50), and `Salary` (Decimal). Then add a new column `Department` (Variable character up to 30) and modify the `Salary` column.

Step 1: Create the Employee table.

```
CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
```

```
    Salary DECIMAL(10, 2)
);
```

Step 2: Add the Department column.

```
ALTER TABLE Employee ADD Department VARCHAR(30);
```

Step 3: Modify the Salary data type.

```
ALTER TABLE Employee MODIFY Salary DECIMAL(12, 2);
```

3.2 Data Manipulation Language (DML) Commands

Methodology:

Inserting Updating and Deleting Records Data Manipulation Language (DML) commands handle data operations within tables.

1. **INSERT INTO:** Adds new rows to a table.

```
INSERT INTO table_name (column1, column2)
VALUES (value1, value2);
```

2. **UPDATE:** Modifies existing rows in a table. Always use a WHERE clause to avoid updating all rows.

```
UPDATE table_name
SET column1 = value1
WHERE condition;
```

3. **DELETE:** Removes specific rows from a table based on a condition.

```
DELETE FROM table_name
WHERE condition;
```

Worked Example:

Performing DML Operations on the Employee Table Insert two records into the `Employee` table. Then update the salary of the employee with `EmpID` 101 and delete the employee with `EmpID` 102.

Step 1: Insert records.

```
INSERT INTO Employee (EmpID, Name, Salary, Department)
VALUES (101, 'Alice Smith', 50000.00, 'IT');

INSERT INTO Employee (EmpID, Name, Salary, Department)
VALUES (102, 'Bob Jones', 45000.00, 'HR');
```

Step 2: Update a record.

```
UPDATE Employee
SET Salary = 55000.00
WHERE EmpID = 101;
```

Step 3: Delete a record.

```
DELETE FROM Employee
WHERE EmpID = 102;
```

3.3 Data Query Language (DQL) Commands

Methodology:

Retrieving Data using SELECT The SELECT statement is used to fetch data from the database.

1. **Basic Retrieval:** Select all or specific columns.

```
SELECT * FROM table_name;
SELECT column1, column2 FROM table_name;
```

2. **Filtering (WHERE):** Retrieve rows that satisfy a condition.

```
SELECT columns FROM table_name WHERE condition;
```

3. **Sorting (ORDER BY):** Sort the result set ascending (ASC) or descending (DESC).

```
SELECT columns FROM table_name ORDER BY column1 ASC;
```

4. **Pattern Matching (LIKE):** Search for a specified pattern in a column using % (zero or more characters) or _ (single character).

```
SELECT columns FROM table_name WHERE column1 LIKE 'A%';
```

Worked Example:

Querying Employee Records Write SQL queries to: 1. Retrieve names and salaries of all employees in the 'IT' department. 2. Retrieve all employees whose name starts with 'S'. 3. Display the list of employees sorted by Salary in descending order.

Solution 1: Filter by department.

```
SELECT Name, Salary
FROM Employee
WHERE Department = 'IT';
```

Solution 2: Pattern matching.

```
SELECT *
FROM Employee
WHERE Name LIKE 'S%';
```

Solution 3: Sort by salary.

```
SELECT *
FROM Employee
ORDER BY Salary DESC;
```

3.4 Aggregate Functions and Grouping

Methodology:

Using GROUP BY and HAVING Aggregate functions perform calculations on multiple rows and return a single value. Common functions include `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()`.

1. **GROUP BY:** Groups rows that have the same values into summary rows.

```
SELECT column1, AGG_FUNC(column2)
FROM table_name
GROUP BY column1;
```

2. **HAVING:** Applies conditions to grouped records (similar to WHERE, but for aggregate functions).

```
SELECT column1, AGG_FUNC(column2)
FROM table_name
GROUP BY column1
HAVING AGG_FUNC(column2) > value;
```

Worked Example:

Calculating Department Statistics Given the `Employee` table, write queries to find the total number of employees in each department, and find departments where the average salary is greater than 60000.
Step 1: Group by department and count employees.

```
SELECT Department, COUNT(EmpID) AS TotalEmployees
FROM Employee
GROUP BY Department;
```

Step 2: Use HAVING to filter grouped data based on average salary.

```
SELECT Department, AVG(Salary) AS AverageSalary
FROM Employee
GROUP BY Department
HAVING AVG(Salary) > 60000;
```

3.5 Joining Tables

Methodology:

SQL Joins 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.

```
SELECT columns FROM TableA
INNER JOIN TableB ON TableA.key = TableB.key;
```

2. **LEFT JOIN:** Returns all records from the left table, and matching records from the right table. Result is NULL on the right side if there is no match.

```
SELECT columns FROM TableA
LEFT JOIN TableB ON TableA.key = TableB.key;
```

3. **RIGHT JOIN:** Returns all records from the right table, and matching 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:

Querying Data Across Multiple Tables Consider two tables: `Employee(EmpID, Name, DeptID)` and `Department(DeptID, DeptName)`. Write a query to display employee names along with their respective department names. Include employees who might not have an assigned department.
Solution: Use a LEFT JOIN. To display all employees and their department names (if assigned), the `Employee` table should be on the left.

```
SELECT Employee.Name, Department.DeptName
FROM Employee
LEFT JOIN Department ON Employee.DeptID = Department.DeptID;
```

3.6 Subqueries

Methodology:

Writing Subqueries A subquery (or inner query) is a query nested inside another SQL query (the outer query). Subqueries can be used with SELECT, INSERT, UPDATE, or DELETE statements.

1. **Single-row Subquery:** Returns zero or one row. Used with single-value operators like `=`, `>`, `<`.
2. **Multi-row Subquery:** Returns one or more rows. Used with operators like `IN`, `ANY`, or `ALL`.

Worked Example:

Filtering using a Subquery Find the names of all employees whose salary is greater than the average salary of all employees in the company.
Step 1: Identify the inner query. Calculate the average salary across all employees.

```
SELECT AVG(Salary) FROM Employee
```

Step 2: Incorporate into the outer query. Select the names of employees whose salary is strictly greater than the result of the inner query.

```
SELECT Name, Salary
FROM Employee
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employee
);
```

CHAPTER 4

Refining Database Design through Normalization

4.1 Functional Dependencies and Attribute Closure

Normalization is a systematic approach to decomposing tables to eliminate data redundancy and undesirable characteristics like insertion, update, and deletion anomalies. The foundation of this process is the concept of Functional Dependencies (FDs). A functional dependency $X \to Y$ implies that the attributes in X uniquely determine the attributes in Y.

Methodology:

Computing Attribute Closure To find the closure of an attribute set X (denoted as X^+) under a set of Functional Dependencies F:

1. Initialize the result set to the given attribute set: $X^+ = X$.
2. Loop through each functional dependency $Y \to Z$ in F.
3. If the left-hand side Y is a subset of the current X^+ ($Y \subseteq X^+$), add the right-hand side Z to X^+ ($X^+ = X^+ \cup Z$).
4. Repeat steps 2 and 3 until the set X^+ stops growing.

Worked Example:

Find the Attribute Closure Given a relation $R(ABCDEF)$ and a set of FDs $F = \{A \to B, B \to C, C \to D, E \to F\}$. Find the closure of the attribute set $\{A, E\}$.
Step 1: Initialize the closure. $\{A, E\}^+ = \{A, E\}$
Step 2: Iterate through the given FDs.

- Check $A \to B$: Since $A \subseteq \{A, E\}^+$, add B. Now, $\{A, E\}^+ = \{A, B, E\}$.
- Check $B \to C$: Since $B \subseteq \{A, E\}^+$, add C. Now, $\{A, E\}^+ = \{A, B, C, E\}$.
- Check $C \to D$: Since $C \subseteq \{A, E\}^+$, add D. Now, $\{A, E\}^+ = \{A, B, C, D, E\}$.
- Check $E \to F$: Since $E \subseteq \{A, E\}^+$, add F. Now, $\{A, E\}^+ = \{A, B, C, D, E, F\}$.

Step 3: Final Result. $\{A, E\}^+ = \{A, B, C, D, E, F\}$. Since the closure contains all attributes of relation R, the set $\{A, E\}$ uniquely determines all attributes in the relation.

Methodology:

Finding Candidate Keys To systematically determine all candidate keys for a relation R given FDs F:

1. Identify all attributes that never appear on the right-hand side (RHS) of any FD. These *essential*

attributes must be a part of every candidate key.

2. Identify all attributes that appear on both sides or only on the RHS.
3. Compute the attribute closure of the essential attributes. If it yields all attributes of R, the essential attributes alone form the only candidate key.
4. If the essential attributes do not yield all of R, iteratively add other attributes (found in step 2) one by one and compute their closures until you find minimal sets that yield all attributes of R.

Worked Example:

Find the Candidate Keys Given a relation $R(ABCD)$ and FDs $F = \{AB \to C, C \to D, D \to A\}$. Find all candidate keys.
Step 1: Identify essential attributes.

- Attributes on RHS: $\{C, D, A\}$.
- Attributes NOT on RHS: $\{B\}$.
- Therefore, B must be present in every candidate key.

Step 2: Compute closure of essential attributes. $\{B\}^+ = \{B\}$. This does not cover all attributes.
Step 3: Test combinations with other attributes. We must combine B with combinations of $\{A, C, D\}$.

- Test $\{A, B\}$: $\{A, B\}^+ \to$ Using $AB \to C$ gives $\{A, B, C\}$. Using $C \to D$ gives $\{A, B, C, D\}$. Since it covers R, AB **is a candidate key**.
- Test $\{B, C\}$: $\{B, C\}^+ \to$ Using $C \to D$ gives $\{B, C, D\}$. Using $D \to A$ gives $\{A, B, C, D\}$. Since it covers R, BC **is a candidate key**.
- Test $\{B, D\}$: $\{B, D\}^+ \to$ Using $D \to A$ gives $\{A, B, D\}$. Using $AB \to C$ gives $\{A, B, C, D\}$. Since it covers R, BD **is a candidate key**.

Final Result: The candidate keys are AB, BC, and BD.

4.2 The Normalization Process

4.2.1 First Normal Form

A relation is in First Normal Form (1NF) if every attribute holds an atomic (indivisible) value. Multi-valued attributes or repeating groups are not allowed.

Methodology:

Converting to First Normal Form To convert an unnormalized table into 1NF:

1. Identify the attributes containing multiple values (repeating groups).
2. Create a new row for each item in the repeating group, duplicating the non-repeating data for those rows.
3. Ensure that every cell contains exactly one value.

Worked Example:

Normalize to 1NF Consider an unnormalized table mapping students to the subjects they study.

StudentID	Name	Subjects
101	Alice	DBMS, OS, Java
102	Bob	OS, CN

Step 1: Identify multi-valued attributes. The `Subjects` column contains multiple values separated by commas.
Step 2 & 3: Create individual rows for each subject while duplicating the `StudentID` and `Name`.

StudentID	Name	Subject
101	Alice	DBMS
101	Alice	OS
101	Alice	Java
102	Bob	OS
102	Bob	CN

The relation is now in 1NF.

4.2.2 Second Normal Form

A relation is in Second Normal Form (2NF) if it is in 1NF and contains no partial dependencies. A partial dependency exists when a non-prime attribute (an attribute not part of any candidate key) is dependent on only a proper subset of a candidate key.

Methodology:

Converting to Second Normal Form To convert a 1NF relation to 2NF:

1. Identify the candidate keys of the relation.
2. Identify any partial dependencies: $X \to Y$ where X is a proper subset of a candidate key and Y is a non-prime attribute.
3. Remove the partial dependency by placing X and Y in a new relation where X serves as the primary key.
4. Keep X in the original relation to serve as a foreign key to link the relations back together.

Worked Example:

Normalize to 2NF Consider relation $R(OrderID, ProductID, ProductName, Quantity)$ with the candidate key $\{OrderID, ProductID\}$.
The FDs are:

- $\{OrderID, ProductID\} \to Quantity$
- $ProductID \to ProductName$

Step 1: The candidate key is $\{OrderID, ProductID\}$. Prime attributes are $OrderID$ and $ProductID$. Non-prime attributes are $ProductName$ and $Quantity$.
Step 2: Identify partial dependencies. $ProductID \to ProductName$ is a partial dependency because $ProductName$ depends only on $ProductID$, which is a proper subset of the candidate key.
Step 3 & 4: Decompose to remove the partial dependency.

- Create a new relation for the partial dependency:
  $R_1(\underline{ProductID}, ProductName)$.
- The original relation retains the full key and the attributes that depend on the full key:
  $R_2(\underline{OrderID, ProductID}, Quantity)$.

The relations R_1 and R_2 are now in 2NF.

4.2.3 Third Normal Form

A relation is in Third Normal Form (3NF) if it is in 2NF and contains no transitive dependencies. A transitive dependency exists when a non-prime attribute depends on another non-prime attribute. Formally, for every non-trivial FD $X \to Y$, either X is a superkey, or Y is a prime attribute.

Methodology:

Converting to Third Normal Form To convert a 2NF relation to 3NF:

1. Identify all non-trivial functional dependencies in the relation.
2. Find any transitive dependencies: $X \to Y$ where X is not a superkey and Y is not a prime attribute.
3. Remove the transitive dependency by creating a new relation containing X and Y, with X as the primary key.
4. Leave X in the original relation to act as a foreign key.

Worked Example:

Normalize to 3NF Consider relation $R(EmployeeID, Name, DeptID, DeptName)$ in 2NF.
The FDs are:

- $EmployeeID \to Name, DeptID$
- $DeptID \to DeptName$

Step 1: The candidate key is $EmployeeID$.
Step 2: Identify transitive dependencies. The FD $DeptID \to DeptName$ violates 3NF because $DeptID$ is not a superkey, and $DeptName$ is a non-prime attribute. This creates a transitive dependency: $EmployeeID \to DeptID \to DeptName$.
Step 3 & 4: Decompose to remove the transitive dependency.

- Create a new relation for the transitive dependency:
  $R_1(\underline{DeptID}, DeptName)$.
- The original relation drops the transitively dependent attribute:
  $R_2(\underline{EmployeeID}, Name, DeptID)$.

The relations R_1 and R_2 are now in 3NF.

4.2.4 Boyce Codd Normal Form

Boyce-Codd Normal Form (BCNF) is a stricter version of 3NF. A relation is in BCNF if, for every non-trivial functional dependency $X \to Y$, X is a superkey. It eliminates redundancies that can still exist in 3NF when overlapping candidate keys exist.

Methodology:

Converting to Boyce Codd Normal Form To convert a relation to BCNF:

1. Identify all non-trivial functional dependencies $X \to Y$.
2. Check if X is a superkey (i.e., X^+ contains all attributes of the relation).
3. If a violation exists (X is not a superkey), decompose the relation into two new relations:
   - One containing $X \cup Y$ (with X as the primary key).
   - Another containing X and the remaining attributes not in Y.

Worked Example:

Normalize to BCNF Consider relation $R(Student, Course, Instructor)$ mapping students to courses and the specific instructor teaching them. A student can take multiple courses, but each course is taught by multiple instructors, and an instructor teaches only one course.
The FDs are:

- $\{Student, Course\} \to Instructor$
- $Instructor \to Course$

Step 1: Determine candidate keys. Closure of $\{Student, Course\}^+$ yields all attributes. Closure of $\{Student, Instructor\}^+$ yields all attributes. Both are candidate keys. Relation is in 3NF (Course is a prime attribute, so $Instructor \to Course$ doesn't violate 3NF).
Step 2: Check BCNF condition. Look at $Instructor \to Course$. Is $Instructor$ a superkey? No, $\{Instructor\}^+ = \{Instructor, Course\}$. It doesn't yield $Student$. This violates BCNF.
Step 3: Decompose based on the violating FD $Instructor \to Course$ (where $X = Instructor, Y = Course$).

- Relation 1: $X \cup Y \to R_1(\underline{Instructor}, Course)$.
- Relation 2: X and remaining attributes $\to R_2(\underline{Student, Instructor})$.

The relations R_1 and R_2 are now in BCNF.

4.3 Properties of Decomposition

When decomposing relations, we must ensure we don't lose data or lose constraints. The two primary properties to verify are Lossless Join and Dependency Preservation.

Methodology:

Checking for Lossless Join Given a relation R decomposed into two relations R_1 and R_2 under a set of FDs F. The decomposition is lossless if the intersection of R_1 and R_2 is a candidate key for either R_1 or R_2. Mathematically, at least one of the following must hold in F^+:

1. $(R_1 \cap R_2) \to R_1$
2. $(R_1 \cap R_2) \to R_2$

If neither holds, the decomposition is lossy, meaning spurious tuples will be generated if R_1 and R_2 are joined.

Worked Example:

Test Lossless Join Given relation $R(ABC)$ and FDs $F = \{A \to B\}$. Let R be decomposed into $R_1(AB)$ and $R_2(AC)$. Determine if the decomposition is lossless.
Step 1: Find the intersection of the decomposed relations. $R_1 \cap R_2 = \{A, B\} \cap \{A, C\} = \{A\}$.
Step 2: Check if the intersection determines either R_1 or R_2.

- Does $A \to R_1$? Since $R_1 = \{A, B\}$, we need to check if $A \to AB$. We know $A \to B$ is in F. Therefore, A is a key for R_1.

Since $\{A\} \to R_1$ is valid, the decomposition is **Lossless**.

Methodology:

Checking Dependency Preservation Given a relation R with FDs F decomposed into $R_1, R_2, \ldots, R_n$. To check if the decomposition is dependency-preserving:

1. Identify the set of FDs F_i that apply to each decomposed relation R_i. (An FD $X \to Y$ applies to R_i if both X and Y are attributes within R_i).
2. Construct a combined set of preserved FDs: $F' = F_1 \cup F_2 \cup \cdots \cup F_n$.
3. For every original FD $X \to Y$ in F, compute the closure X^+ using **only** the FDs in F'.
4. If $Y \subseteq X^+$ for all FDs in F, the decomposition preserves dependencies. If even one dependency cannot be derived using F', it is not dependency-preserving.

Worked Example:

Test Dependency Preservation Given relation $R(City, Street, ZipCode)$ and FDs $F = \{\{City, Street\} \to ZipCode, ZipCode \to City\}$. The relation is decomposed into $R_1(ZipCode, City)$ and $R_2(Street, ZipCode)$. Check if this is dependency-preserving.

Step 1: Find dependencies applicable to each new relation.

- In $R_1(ZipCode, City)$, the FD $ZipCode \to City$ applies. So, $F_1 = \{ZipCode \to City\}$.
- In $R_2(Street, ZipCode)$, neither of the original FDs apply directly (since $\{City, Street\}$ requires attributes from both tables). So, $F_2 = \emptyset$.

Step 2: Form the combined preserved FDs. $F' = F_1 \cup F_2 = \{ZipCode \to City\}$.

Step 3: Test each original FD using only F'.

- Test $ZipCode \to City$: Under F', $\{ZipCode\}^+ = \{ZipCode, City\}$. The dependency holds.
- Test $\{City, Street\} \to ZipCode$: Under F', the closure of $\{City, Street\}^+$ gives $\{City, Street\}$. It does not include $ZipCode$.

Step 4: Conclusion. Since $\{City, Street\} \to ZipCode$ cannot be derived from the decomposed schemas, the decomposition is **Not Dependency-Preserving**.

CHAPTER 5

Transaction Management

5.1 Testing for Conflict Serializability

Methodology:

Precedence Graph Method To determine if a given concurrent schedule S of transactions is conflict serializable, construct a precedence graph (also known as a serialization graph).

1. **Create Nodes:** For each transaction T_i in the schedule S, create a node labeled T_i.
2. **Identify Conflicting Operations:** Two operations conflict if they belong to different transactions, access the same data item, and at least one is a write operation. The three types of conflicts are:
   - Read-Write (RW) conflict
   - Write-Read (WR) conflict
   - Write-Write (WW) conflict
3. **Draw Directed Edges:** For every conflicting pair of operations where an operation of T_i executes before an operation of T_j, draw a directed edge from T_i to T_j ($T_i \to T_j$).
4. **Check for Cycles:** Analyze the resulting directed graph.
   - If the graph contains **no cycles**, the schedule is conflict serializable.
   - If the graph contains **one or more cycles**, the schedule is not conflict serializable.
5. **Equivalent Serial Schedule:** If acyclic, perform a topological sort on the graph to find the equivalent serial schedule(s).

Worked Example:

Checking Conflict Serializability Determine whether the following schedule S is conflict serializable. If it is, find the equivalent serial schedule.

$$S : R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B)$$

Solution:

1. **Identify Transactions:** The schedule contains two transactions, T_1 and T_2.
2. **List Conflicting Operations:** We look for operations on the same data item (A or B) by different transactions where at least one is a write.
   - Data item A:
     - $W_1(A)$ is followed by $R_2(A) \implies$ Conflict $T_1 \to T_2$
     - $W_1(A)$ is followed by $W_2(A) \implies$ Conflict $T_1 \to T_2$
     - $R_1(A)$ is followed by $W_2(A) \implies$ Conflict $T_1 \to T_2$

- Data item B:
  - $W_1(B)$ is followed by $R_2(B) \implies$ Conflict $T_1 \to T_2$
  - $W_1(B)$ is followed by $W_2(B) \implies$ Conflict $T_1 \to T_2$
  - $R_1(B)$ is followed by $W_2(B) \implies$ Conflict $T_1 \to T_2$

3. **Construct Precedence Graph:** Based on the conflicts, we draw directed edges. All conflicts indicate an edge from T_1 to T_2. There are no edges from T_2 to T_1.
4. **Check for Cycles:** The graph consists of nodes T_1 and T_2 with a single directed edge $T_1 \to T_2$. There are no cycles.
5. **Conclusion:** Since the precedence graph is acyclic, schedule S is conflict serializable. The topological sort yields the equivalent serial schedule: $T_1 \to T_2$.

Worked Example:

Schedule with a Cycle Determine if the following schedule S is conflict serializable.

$$S : R_1(X), R_2(X), W_1(X), W_2(X)$$

Solution:

1. **Identify Transactions:** T_1 and T_2.
2. **List Conflicting Operations on X:**
   - $R_2(X)$ is followed by $W_1(X) \implies$ Conflict $T_2 \to T_1$
   - $R_1(X)$ is followed by $W_2(X) \implies$ Conflict $T_1 \to T_2$
   - $W_1(X)$ is followed by $W_2(X) \implies$ Conflict $T_1 \to T_2$
3. **Construct Precedence Graph:** We have edges $T_1 \to T_2$ and $T_2 \to T_1$.
4. **Check for Cycles:** The edges form a cycle $T_1 \leftrightarrow T_2$.
5. **Conclusion:** Because a cycle exists, the schedule S is **not** conflict serializable.

5.2 Testing for View Serializability

Methodology:

View Equivalence Conditions Two schedules S_1 and S_2 on the same set of transactions are view equivalent if they satisfy three conditions for every data item Q:

1. **Initial Read:** If transaction T_i reads the initial value of Q in S_1, then T_i must also read the initial value of Q in S_2.
2. **Update Read:** If transaction T_i executes $R_i(Q)$ in S_1 and the value was produced by $W_j(Q)$ (from transaction T_j), then T_i must also read the value produced by $W_j(Q)$ in S_2.
3. **Final Write:** If transaction T_k performs the final write operation $W_k(Q)$ in S_1, then T_k must also perform the final write $W_k(Q)$ in S_2.

A schedule is view serializable if it is view equivalent to any serial schedule. (Note: Every conflict serializable schedule is also view serializable. The reverse is not always true, specifically when there are "blind writes".)

Worked Example:

Checking View Serializability Consider the schedule $S : R_1(X), W_2(X), W_1(X), W_3(X)$. Determine if S is view serializable.
Solution:

1. **Identify Transactions:** T_1, T_2, T_3.
2. **Analyze Schedule S for Data Item X:**
   - **Initial Read:** T_1 reads the initial value of X $(R_1(X))$.
   - **Final Write:** T_3 performs the final write on X $(W_3(X))$.
   - **Update Read:** There are no reads of updated values. $W_2(X)$ and $W_1(X)$ are blind writes.
3. **Test Serial Schedules:** We must find a serial schedule of T_1, T_2, T_3 that meets the conditions.
   - To satisfy the **Initial Read** condition, T_1 must execute before any transaction writes to X. Thus, T_1 must be the first transaction.
   - To satisfy the **Final Write** condition, T_3 must execute last.
   - The only serial schedule that starts with T_1 and ends with T_3 is $S_{serial} : T_1 \to T_2 \to T_3$.
4. **Verify Equivalence:** Let's check $S_{serial} = R_1(X), W_1(X), W_2(X), W_3(X)$ against S:
   - Initial read in S_{serial}: T_1 reads initial X. (Matches S)
   - Final write in S_{serial}: T_3 writes last X. (Matches S)
   - Update read: None in S or S_{serial}. (Matches S)
5. **Conclusion:** Schedule S is view equivalent to the serial schedule $T_1 \to T_2 \to T_3$. Therefore, S is view serializable.

5.3 Concurrency Control using Timestamp Ordering

Methodology:

Timestamp Ordering Protocol Each transaction T_i is assigned a unique timestamp $TS(T_i)$ when it begins. Each data item Q maintains two timestamp values:

- $W\text{-}TS(Q)$: Largest timestamp of any transaction that successfully executed $W(Q)$.
- $R\text{-}TS(Q)$: Largest timestamp of any transaction that successfully executed $R(Q)$.

The protocol ensures serializability by evaluating operations:

1. **Transaction T_i issues $R_i(Q)$:**
   - If $TS(T_i) < W\text{-}TS(Q)$: T_i needs to read a value that was already overwritten. **Reject** $R_i(Q)$ and rollback T_i.
   - If $TS(T_i) \geq W\text{-}TS(Q)$: **Accept** $R_i(Q)$. Update $R\text{-}TS(Q) = \max(R\text{-}TS(Q), TS(T_i))$.
2. **Transaction T_i issues $W_i(Q)$ (Basic Protocol):**
   - If $TS(T_i) < R\text{-}TS(Q)$: The value T_i is producing was needed previously. **Reject** $W_i(Q)$ and rollback T_i.
   - If $TS(T_i) < W\text{-}TS(Q)$: T_i is trying to write an obsolete value. **Reject** $W_i(Q)$ and rollback T_i.
   - Otherwise: **Accept** $W_i(Q)$. Update $W\text{-}TS(Q) = TS(T_i)$.
3. **Thomas Write Rule (Modification for Writes):**

- If $TS(T_i) < R\text{-}TS(Q)$: **Reject** $W_i(Q)$ and rollback T_i.
- If $TS(T_i) < W\text{-}TS(Q)$: **Ignore** the write operation (do not rollback, just skip the write). This is a safe blind write.
- Otherwise: **Accept** $W_i(Q)$. Update $W\text{-}TS(Q) = TS(T_i)$.

Worked Example:

Applying Timestamp Ordering Consider three transactions with timestamps $TS(T_1) = 10$, $TS(T_2) = 20$, and $TS(T_3) = 30$. Initially, $R\text{-}TS(X) = 0$ and $W\text{-}TS(X) = 0$. Determine if the following sequence of operations is accepted under the Basic Timestamp Ordering Protocol. Sequence: $R_1(X), W_2(X), R_3(X), W_1(X)$
Solution: We evaluate each operation step-by-step and track the timestamps of data item X.

1. **Operation $R_1(X)$:**
   - Compare $TS(T_1)$ to $W\text{-}TS(X)$: $10 \geq 0$.
   - Action: **Accept**.
   - Update: $R\text{-}TS(X) = \max(0, 10) = 10$.
2. **Operation $W_2(X)$:**
   - Compare $TS(T_2)$ to $R\text{-}TS(X)$ and $W\text{-}TS(X)$.
   - $20 \geq 10$ and $20 \geq 0$.
   - Action: **Accept**.
   - Update: $W\text{-}TS(X) = 20$.
3. **Operation $R_3(X)$:**
   - Compare $TS(T_3)$ to $W\text{-}TS(X)$: $30 \geq 20$.
   - Action: **Accept**.
   - Update: $R\text{-}TS(X) = \max(10, 30) = 30$.
4. **Operation $W_1(X)$:**
   - Compare $TS(T_1)$ to $R\text{-}TS(X)$: $10 < 30$.
   - Action: **Reject**. Transaction T_1 must be rolled back.

5.4 Deadlock Detection

Methodology:

Wait-For Graph Method A Wait-For graph is used to detect deadlocks in a system using lock-based concurrency control.

1. **Create Nodes:** Create a node for each active transaction T_i.
2. **Draw Edges:** Draw a directed edge from T_i to T_j ($T_i \rightarrow T_j$) if transaction T_i is waiting to acquire a lock on a data item that is currently held by transaction T_j.
3. **Detect Cycle:** Periodically check the graph for cycles.
   - If a cycle exists, the system is in a **deadlock** state. The transactions in the cycle are deadlocked.
   - To recover, select a "victim" transaction within the cycle and abort it.

Worked Example:

Constructing a Wait-For Graph Four transactions T_1, T_2, T_3, T_4 execute concurrently. Based on the following lock requests and assignments, determine if a deadlock exists.

- T_1 holds an exclusive lock on A.
- T_2 holds a shared lock on B.
- T_3 holds an exclusive lock on C.
- T_1 requests an exclusive lock on B.
- T_4 requests an exclusive lock on C.
- T_2 requests a shared lock on A.
- T_3 requests an exclusive lock on A.

Solution:

1. **Nodes:** T_1, T_2, T_3, T_4.
2. **Determine Wait Conditions (Edges):**
   - T_1 requests lock on B. B is held by T_2. $\implies$ Edge $T_1 \to T_2$.
   - T_4 requests lock on C. C is held by T_3. $\implies$ Edge $T_4 \to T_3$.
   - T_2 requests lock on A. A is held by T_1. $\implies$ Edge $T_2 \to T_1$.
   - T_3 requests lock on A. A is held by T_1. $\implies$ Edge $T_3 \to T_1$.
3. **Analyze the Graph:** The Wait-For graph has edges: $T_1 \to T_2$, $T_2 \to T_1$, $T_4 \to T_3$, and $T_3 \to T_1$.
4. **Check for Cycles:** We can clearly see a cycle involving T_1 and T_2: $T_1 \to T_2 \to T_1$.
5. **Conclusion:** The system is in a **deadlock** involving transactions T_1 and T_2.

5.5 Log-Based Recovery

Methodology:

Deferred and Immediate Modification Rules When a system crash occurs, the recovery manager reads the log file to restore the database to a consistent state. A log record has the format $\langle T_i, X, \text{old\_val}, \text{new\_val} \rangle$ (for immediate modification).

Key Concepts:

- $\langle T_i, \text{start} \rangle$: Transaction T_i began.
- $\langle T_i, \text{commit} \rangle$: Transaction T_i successfully completed.
- $\langle \text{checkpoint} \rangle$: All modified buffers up to this point were safely written to disk.

Immediate Database Modification Method:

1. **Scan Log:** Scan the log file backwards from the end to the most recent $\langle \text{checkpoint} \rangle$. Determine the lists of Undo and Redo transactions.
2. **Redo List:** A transaction T_i needs to be REDONE if the log contains both $\langle T_i, \text{start} \rangle$ and $\langle T_i, \text{commit} \rangle$.
3. **Undo List:** A transaction T_i needs to be UNDONE if the log contains $\langle T_i, \text{start} \rangle$ but **does not** contain $\langle T_i, \text{commit} \rangle$.

4. **Execute Undo:** Scan backward from the end. For each log record of a transaction in the Undo list, restore the data item to its **old_val**.
5. **Execute Redo:** Scan forward from the checkpoint. For each log record of a transaction in the Redo list, set the data item to its **new_val**.

Deferred Database Modification Method: Differences from Immediate:

- Updates are not written to the actual database until commit. Log records only store new values: $\langle T_i, X, \text{new\_val} \rangle$.
- **No Undo phase** is required because uncommitted transactions never modified the database.
- Only the **Redo phase** is executed for committed transactions.

Worked Example:

Immediate Database Modification Recovery A system crashes. The log file just before the crash contains the following entries (in order):

```
1. <T1, start>
2. <T1, A, 100, 200>
3. <T2, start>
4. <T2, B, 50, 150>
5. <T1, commit>
6. <T3, start>
7. <T3, C, 10, 20>
8. <T2, B, 150, 250>
[CRASH]
```

Assume initial values on disk are A=100, B=50, C=10. Determine the Undo list, Redo list, and the final values of A, B, and C after recovery using Immediate Database Modification.
Solution:

1. **Determine Undo and Redo Lists:**
   - T_1 has $\langle T_1, \text{start} \rangle$ and $\langle T_1, \text{commit} \rangle$. $\implies$ **Redo List: { T_1 }**
   - T_2 has $\langle T_2, \text{start} \rangle$ but no commit. $\implies$ **Undo List: { T_2 }**
   - T_3 has $\langle T_3, \text{start} \rangle$ but no commit. $\implies$ **Undo List: { T_3, T_2 }**
2. **Perform Undo Operations:** Scan backward from the crash point for transactions in the Undo List (T_2, T_3). Set items to **old_val**.
   - Record 8: $\langle T_2, B, 150, 250 \rangle \implies$ Undo: $B = 150$.
   - Record 7: $\langle T_3, C, 10, 20 \rangle \implies$ Undo: $C = 10$.
   - Record 4: $\langle T_2, B, 50, 150 \rangle \implies$ Undo: $B = 50$.
3. **Perform Redo Operations:** Scan forward for transactions in the Redo List (T_1). Set items to **new_val**.
   - Record 2: $\langle T_1, A, 100, 200 \rangle \implies$ Redo: $A = 200$.
4. **Final Values on Disk:**
   - $A = 200$ (updated by Redo of T_1)
   - $B = 50$ (restored by Undo of T_2)
   - $C = 10$ (restored by Undo of T_3)

Worked Example:

Checkpointing in Recovery Consider the following log:

```
1. <T1, start>
2. <T1, A, 10, 20>
3. <T1, commit>
4. <checkpoint>
5. <T2, start>
6. <T2, B, 20, 30>
7. <T3, start>
8. <T2, commit>
9. <T3, C, 30, 40>
[CRASH]
```

Determine the recovery actions using Immediate Database Modification.
Solution:

1. **Analyze Checkpoint:** The ⟨checkpoint⟩ at record 4 guarantees that all updates prior to it (i.e., T_1's updates) are safely on disk. We do not need to redo T_1 if we start scanning from the checkpoint.
2. **Determine Undo/Redo Lists (post-checkpoint):**
   - T_2 started after checkpoint and committed before crash. $\implies$ **Redo List: { T_2 }**
   - T_3 started after checkpoint but did not commit. $\implies$ **Undo List: { T_3 }**
3. **Execute Undo (Backward Scan):**
   - Record 9: $\langle T_3, C, 30, 40\rangle \implies$ Undo: $C = 30$.
4. **Execute Redo (Forward Scan from Checkpoint):**
   - Record 6: $\langle T_2, B, 20, 30\rangle \implies$ Redo: $B = 30$.

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book. Specific examples and numeric substitutions have been generalized.

Unit 2: Relational Algebra

Basic Operations

$$\textbf{Selection:} \quad \sigma_P(R) \tag{5.1}$$

$$\textbf{Projection:} \quad \pi_{A_1,A_2,\ldots,A_n}(R) \tag{5.2}$$

$$\textbf{Set Difference:} \quad R_1 - R_2 \tag{5.3}$$

$$\textbf{Union:} \quad R_1 \cup R_2 \tag{5.4}$$

$$\textbf{Intersection:} \quad R_1 \cap R_2 \tag{5.5}$$

$$\textbf{Cartesian Product:} \quad R_1 \times R_2 \tag{5.6}$$

Where R, R_1, R_2 are relations, P is a predicate condition, and $A_1, \ldots, A_n$ are attributes.

Extended Operations

$$\textbf{Natural Join:} \quad R_1 \bowtie R_2 \tag{5.7}$$

$$\textbf{Theta Join:} \quad R_1 \bowtie_\theta R_2 \tag{5.8}$$

Where θ is the join condition.

Unit 4: Relational Database Design

Functional Dependencies

$$\textbf{Attribute Closure:} \quad X^+ \tag{5.9}$$

$$\textbf{Trivial Dependency:} \quad X \to Y \text{ where } Y \subseteq X \tag{5.10}$$

$$\textbf{Dependency Preservation:} \quad F' = F_1 \cup F_2 \cup \cdots \cup F_n \tag{5.11}$$

Where X and Y are sets of attributes, and F' represents the union of functional dependencies preserved across decomposed relations $R_1, R_2, \ldots, R_n$.

Unit 5: Concurrency Control

Timestamp Ordering Protocol

$$\textbf{Read Timestamp Update:} \quad R\text{-}TS(Q) = \max(R\text{-}TS(Q), TS(T_i)) \tag{5.12}$$

$$\textbf{Write Timestamp Update:} \quad W\text{-}TS(Q) = \max(W\text{-}TS(Q), TS(T_i)) \tag{5.13}$$

Where $TS(T_i)$ is the timestamp of transaction T_i, and $R\text{-}TS(Q)$ and $W\text{-}TS(Q)$ are the read and write timestamps of data item Q, respectively.