Unit –IV: RELATIONAL DATABASE DESIGN

Schema refinement And Normal Forms

Schema refinement is the process of organizing data in a database to reduce redundancy and avoid anomalies (problems with data insertion, updating, and deletion). The primary technique used for schema refinement is **normalization**, which involves decomposing large tables into smaller, well-structured tables based on a set of rules called **Normal Forms**.

Schema Refinement

Poorly designed database schemas, where all data is stored in a single large table, can lead to:

Redundancy: The same information is repeated across multiple rows, wasting storage space.

- **Update Anomalies:** Updating a piece of data requires updating every instance of it, leading to inconsistencies if some are missed.
- **Insertion Anomalies:** Cannot insert a new record without having data for all columns, even if some data is not yet available (e.g., adding a new department before any employees are assigned to it).
- **Deletion Anomalies:** Deleting a row might unintentionally remove other related data (e.g., deleting a student's last course enrollment might delete their entire personal record if it's the only one).

Schema refinement uses functional dependencies (FDs) to systematically break down (decompose) the original table into smaller tables that are free of these problems. The goal is to achieve a "lossless-join decomposition" and "dependency preservation" to ensure no information is lost and all original constraints can still be enforced.

Normal Forms (NF) with Examples

Normal forms are a progressive set of rules. A relation in a higher normal form satisfies the conditions of all lower normal forms. The most common forms are 1NF, 2NF, 3NF, and BCNF.

- **First Normal Form (1NF):** A table is in 1NF if every column contains only **atomic (single) values**, and there are no repeating groups of data.

Example Violation: A `Student` table with a `Courses` column storing "C, Java".

1NF Solution: Each course is listed in a separate row, or a new table is created, ensuring each cell has a single value.

- **Second Normal Form (2NF):** A table is in 2NF if it is in 1NF and all non-key attributes are **fully dependent on the primary key**. This means removing partial dependencies where an attribute depends on only a part of a composite primary key.

  Example Violation: In a `(StudentID, CourseID)` primary key table, `StudentName` depends only on `StudentID`.

  2NF Solution: Move `StudentID` and `StudentName` to a separate `Student` table. The original table is left with `(StudentID, CourseID)` and perhaps the `Grade`.

- **Third Normal Form (3NF):** A table is in 3NF if it is in 2NF and has **no transitive dependencies**. A transitive dependency occurs when a non-key attribute is dependent on another non-key attribute (e.g., A → B and B → C implies A → C).

  Example Violation: In a table with `(StudentID, StudentName, ZIP, City, Country)`, `ZIP` determines `City` and `Country`, but `ZIP` is not the key.

  3NF Solution: Decompose into a `Student (StudentID, StudentName, and ZIP)` table and a `ZIP_Info (ZIP, City, and Country)` table.

- **Boyce-Codd Normal Form (BCNF):** This is a stricter version of 3NF. A table is in BCNF if, for every non-trivial functional dependency X → Y, **X must be a super key** (meaning X determines all other attributes in the table). BCNF handles some anomalies that 3NF might miss in specific scenarios involving multiple overlapping candidate keys.

  - **Example Violation:** In a `(StudentID, Course, Instructor)` table, if `Course` determines `Instructor`, but neither `Course` nor `Instructor` alone is a super key.

  - **BCNF Solution:** Decompose into `Student_Course (StudentID, Course)` and `Course_Instructor (Course, Instructor)` tables.

Introduction to schema refinement

Schema refinement in a Database Management System (DBMS) is the structured process of analyzing and transforming an initial database design into a higher quality, optimized structure using a method called **normalization**. The primary goal is to minimize data redundancy and prevent data anomalies (issues with inserting, updating, or deleting data consistently).

The Problem with an Unrefined Schema

Consider an initial, unrefined table called `EmployeeProject` that manages employee details, their department, and the projects they work on.

EmpID	EmpName	DeptName	DeptPhone	ProjectID	ProjectName	Hours
101	Srikanth	Sales	9765-4386	P1	Alpha	40
101	Srikanth	Sales	9765-4386	P2	Beta	20
102	John	IT	9865-2543	P1	Alpha	30

This single table exhibits several problems:

- **Redundancy:** Employee name and department details (`DeptName`, `DeptPhone`) are repeated for every project the employee is on.
- **Update Anomalies:** If Srikanth changes their department phone number, we must update two rows consistently. If one is missed, the data becomes inconsistent.
- **Insertion Anomalies:** We cannot add a new department until an employee is assigned to it.
- **Deletion Anomalies:** If we delete the last row for Employee John (perhaps they are briefly unassigned), we lose all information about Project Alpha's name and ID entirely.

Schema Refinement via Normalization (Example in 3NF)

To refine this schema, we use normalization, breaking the large table into smaller, logically related tables based on functional dependencies (e.g., `EmpID` determines `EmpName`, `DeptID` determines `DeptPhone`). The goal is often to achieve **Third Normal Form (3NF)** or Boyce-Codd Normal Form (BCNF).

Here is how we decompose the original table into three refined tables in 3NF:

1. `Employee` Table

This table stores stable employee-to-department information.

`EmployeeDetails` (EmpID, EmpName, DeptName, DeptPhone)

EmpID	EmpName	DeptName	DeptPhone
101	Srikanth	Sales	555-1234
102	John	IT	555-5678

2. `Project` Table

This table stores project details.`Project` (ProjectID, ProjectName)

ProjectID	ProjectName
P1	Alpha
P2	Beta

3. `WorksOn` (Junction/Relationship) Table

This table records which employees work on which projects and for how many hours, acting as a link between the other two tables. This handles the many-to-many relationship.
`WorksOn` (EmpID, ProjectID, Hours)

EmpID	ProjectID	Hours
101	P1	40
101	P2	20
102	P1	30

Benefits of the Refined Schema

By decomposing the schema, the redundancy is eliminated and anomalies are prevented:

- **No Redundancy:** Employee and department data are stored only once in the `EmployeeDetails` table.
- **No Update Anomalies:** Srikanth's phone number can be updated in a single row without risk of inconsistency.

- **No Insertion Anomalies:** We can add a new department into `EmployeeDetails` even if no one is assigned to a project yet. We can add a new project into the `Project` table instantly.
- **No Deletion Anomalies:** Deleting John's work assignment from `WorksOn` does not delete the definition of Project Alpha itself.

Schema refinement is a critical database design step that results in a stable, consistent, and maintainable database structure.

Key Concepts of Schema Refinement (DBMS)

1. The Goal: Eliminating Anomalies and Redundancy

The main objective is to prevent **data anomalies** (insertion, update, and deletion problems) that occur when a database design is inefficient, typically by storing data redundantly in single, large tables.

2. Functional Dependencies (FDs)

Functional Dependencies are the rules governing the data. They describe constraints between attributes and are the engine that drives normalization. The notation

X→Y X right arrow cap Y□→□ means "if you know X, you can determine Y."

Example Scenario: Unrefined `StudentEnrollment` Table

StudentID	StudentName	CourseCode	CourseName	Professor
100	Alice	CS101	Intro to CS	Dr. Smith
100	Alice	MATH202	Calculus II	Dr. Jones
200	Bob	CS101	Intro to CS	Dr. Smith

Identified FDs:
StudentID →StudentName (A student ID determines only one name.)

CourseCode →CourseName, Professor (A course code determines its name and professor.)

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

Normal forms are a series of rules applied sequentially. Each form has stricter criteria than the last, moving the schema closer to an ideal structure.

Normal Form	Rule	Problem Addressed

1NF	Atomic values only (no lists in cells).	Non-atomic attributes.
2NF	In 1NF + remove partial dependencies.	Partial Key Dependencies.
3NF	In 2NF + remove transitive dependencies.	Indirect dependencies.
BCNF	Stricter 3NF (every determinant is a superkey).	Overlapping candidate keys.

4. Decomposition

When a table violates a normal form rule, it must be decomposed (split) into two or more smaller tables.

Applying Decomposition to the Example (Moving to 3NF):

The original table violates 2NF and 3NF because non-key attributes (CourseName, Professor) depend on only a part of the composite key (CourseCode), and one non-key attribute (Professor) depends on another non-key attribute (CourseCode).

We decompose the single table into three refined tables:

Concept	Original Table	Refined Table(s)
FDs	StudentID →StudentName	Students
Decomposition	CourseCode -→ CourseName, Professor	Courses
	Composite Key: (StudentID, CourseCode)	Enrollment

The Refined Schema Example:

Students (StudentID, StudentName)	Courses (CourseCode, CourseName, Professor)	Enrollment (StudentID, CourseCode)
100, Ajay	CS101, Intro to CS, Dr. Srikanth	100, CS101
200, Baskar	MATH202, Calculus II, Dr. Jayaprakash	100, MATH202

5. Lossless-Join Property

A crucial theoretical concept is that any decomposition must be lossless. This guarantees that you can join the new, smaller tables back together and perfectly reconstruct the original data without losing any information or creating "spurious" (incorrect) rows.

In the example above, joining Students, Courses, and Enrollment correctly restores the original data because we used the common keys (StudentID and CourseCode) for the split.

6. Dependency Preservation

This concept ensures that all original constraints (FDs) can still be checked efficiently using the new, smaller tables. In our example:
We can check StudentID →StudentName within the Students table alone.

We can check CourseCode →CourseName, Professor within the Courses table alone.

Functional Dependencies

A functional dependency (FD) in DBMS defines a relationship between two sets of attributes within a relation. It asserts that if you know the value of one set of attributes (called the determinant), you can uniquely determine the value of another set of attributes**.**

Functional dependency is used to resolve two of the major problems in data in a consistent form and remove redundancy problems by stating a relationship among the attributes columns in a relational database. It solves. This is a very basic concept in database normalization: arranging data such that it doesn't repeat and hence makes logical sense for storing.

Functional dependency is a relationship between two sets of attributes in the table of a database. It occurs when one set of attributes' value uniquely determines the value of another set of attributes, called the dependent. In simpler words, if the value of attribute X determines the value of attribute Y, then we say that Y is functionally dependent on X and denote this by X → Y.

. It is a fundamental concept in database design and the basis for normalization.

Definition and Notation

A functional dependency is denoted as **X → Y**, where:

- **X** is the **determinant** (the attribute or set of attributes on the left side of the arrow).
- **Y** is the **dependent** (the attribute or set of attributes on the right side).

The dependency X → Y means that for any two rows in the table, if they have the same value for X, they must also have the same value for Y. In simpler terms, if you know the value of X, you can determine the value of Y.

Example

Consider an `Employee` table with the attributes `EmployeeID`, `EmployeeName`, and `Department`.

EmployeeID	EmployeeName	Department
101	Arun	Sales
102	Baskar	IT
103	Arun	Sales

The following functional dependencies hold true based on typical business rules:

- EmployeeID → EmployeeName: Each EmployeeID corresponds to only one EmployeeName.
- EmployeeID → Department: Each EmployeeID corresponds to only one Department.
- EmployeeID → EmployeeName, Department: A single employee ID determines all other details.

The following functional dependency is *invalid* in this example:

- EmployeeName → Department: "Alice" appears twice, but the department is "Sales" in both cases. However, if another "Alice" worked in "Marketing", this dependency would be invalid across the entire table. The dependency must hold for all possible data that follows the business rules.

2. Types of Functional Dependencies

Functional dependencies are classified based on the relationship between the determinant and the dependent attributes.

Trivial Functional Dependency: Occurs when the dependent attributes(Y) are a subset of the determinant attributes(X)

- **Example:** {EmployeeID, Name} → Name. This is always true and provides no new information.
- **Non-Trivial Functional Dependency:** Occurs when the dependent attributes are not a subset of the determinant.
    - **Example:** EmployeeID → Department.
- **Full Functional Dependency:** Occurs when a non-key attribute is dependent on the entire composite key, but not on any proper subset of that key.
    - **Example:** In an OrderDetails table with a composite key (OrderID, ProductID), (OrderID, ProductID) → Quantity is a full FD because you need both the order and product ID to know the quantity ordered.
- **Partial Functional Dependency:** Occurs when a non-key attribute depends on only a part of a composite key. This type is a source of redundancy and is addressed during normalization to 2NF.
    - **Example:** If (OrderID, ProductID) is the composite key, ProductID → ProductName is a partial FD because the product name depends only on the ProductID.
- **Transitive Functional Dependency:** An indirect dependency that arises when one attribute determines a second, which in turn determines a third (A → B and B → C implies A → C). This is addressed in 3NF.
    - **Example:** If EmployeeID → DepartmentID and DepartmentID → DepartmentName, then EmployeeID → DepartmentName is a transitive dependency.

3. Properties and Inference Rules (Armstrong's Axioms)

To systematically work with and derive all possible functional dependencies from a given set, a set of rules known as **Armstrong's Axioms** are used.

Axiom	Description	Example
Reflexivity	If Y is a subset of X, then X → Y holds.	{EmpID, Name} → EmpID
Augmentation	If X → Y holds, then XZ → YZ also holds for any attribute set Z.	If EmpID → Name, then {EmpID, City} → {Name, City}
Transitivity	If X → Y and Y → Z hold, then X → Z holds.	If EmpID → Dept and Dept → Building, then EmpID → Building

These axioms are considered **sound** (they only generate valid FDs) and **complete** (they can generate all possible valid FDs). Other rules (like Union and Decomposition) are derived from these fundamental axioms.

Advantages of using functional dependencies

Functional dependencies play a significant role in ensuring that the data of a database is reliable and consistent. Here are some of the key advantages of understanding and using functional dependencies:

- **Data Integrity:** Functional dependencies ensure data consistency, so that all the relationships between the attributes are sound in a logical sense. By enforcing functional dependencies, we can eliminate update, insert, and delete anomalies, thus making sure that the data is correct and consistent.

- **Normalization**: Database normalization depends largely on functional dependencies. It forms the core of database normalization. It prevents data redundancy, making the organization more efficient and streamlined. The overall removal of redundancy in a database is possible and enhances storage space and query speed.

- **Database design:** Functional dependencies make a database's structure easier to update or expand. It enables the designer to understand relationships better between the attributes, making designing an optimized, well-structured database easy
- **Query Performance:** Redundancy makes query executions take longer; with functional dependency, it saves more time while doing queries. By eliminating as much redundancy from the data, queries will perform faster as more data are skipped and scanned while improving overall performance.

Reasoning about FDS

Reasoning about functional dependencies involves analyzing the **business rules and logical constraints** that govern the data, not just the data instance at a specific moment in time. This reasoning helps identify all valid dependencies and use them for proper database normalization and design.

Reasoning Process and Example

The process involves identifying which attributes uniquely determine the values of other attributes based on real-world constraints.

Example Scenario

Consider a university database table `Enrollment` with the following attributes:

- `StudentID` (unique ID for each student)
- `StudentName`
- `CourseID`
- `CourseName`
- `Instructor` (who teaches the specific section of the course)
- `InstructorOffice`

Step 1: Understand Business Rules

The business rules (real-world constraints) define the dependencies:

- Each student ID is unique to a student name.
- Each course ID corresponds to a unique course name.
- A specific `StudentID` enrolled in a specific `CourseID` has a single `Instructor` for that course section.
- Each instructor has a unique `InstructorOffice`.

Step 2: Identify and Reason about FDs

Based on the rules, we can identify and reason about the following FDs:

- **`StudentID` → `StudentName`**: This holds because each student ID is unique to one student name. We can determine a student's name if we know their ID.
  - *Reasoning:* No two students can share the same `StudentID` and have different names.
- **`CourseID` → `CourseName`**: This holds because each course ID has only one name.

- *Reasoning:* A course ID like "CS101" will always be "Intro to Programming".

- **{StudentID, CourseID}** → **Instructor**: This holds because the combination of a student and a course determines the specific instructor teaching that section.
  - *Reasoning:* John enrolled in CS101 has one specific instructor, Professor Sujith.
- **Instructor** → **InstructorOffice**: This holds because each instructor has only one office.
  - *Reasoning:* If we know the instructor is Professor Sujith, we know their office number is B201.

Step 3: Identify Invalid FDs (and potential anomalies)

Reasoning also helps identify FDs that do not hold, which often indicates design flaws:

- **StudentName** → **StudentID** (Invalid): Multiple students could have the same name ("Jaya Surya"), but they will have different unique IDs.
- **Instructor** → **StudentName** (Invalid): An instructor teaches many students. Knowing the instructor doesn't uniquely identify a single student name.
- **StudentID** → **InstructorOffice** (Invalid): This is an indirect dependency (StudentID → CourseID → Instructor → InstructorOffice) and leads to redundancy. The office information would be repeated for every course an instructor teaches. This is a **transitive dependency** that should be removed during normalization.

Importance of Reasoning

By reasoning about these dependencies, a database designer can:

- **Normalize the schema:** Break down the large Enrollment table into smaller, well-structured tables (e.g., Students, Courses, Instructors, Enrollments) to eliminate redundancy and anomalies.
- **Ensure data integrity:** Define primary and foreign keys that enforce these FDs, ensuring consistent and accurate data storage.
- **Improve performance:** Smaller, focused tables with fewer redundancies often lead to faster query execution.

Normal Forms

Normalization is a systematic process of organizing data in a database to reduce data redundancy and eliminate anomalies (insertion, update, and deletion anomalies). It involves applying a set of rules called **Normal Forms** (NF) to decompose large tables into smaller, related tables.

The most common normal forms are First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF).

First Normal Form (1NF)

A table is in 1NF if every column contains only **atomic (single, indivisible) values**, and there are no repeating groups of data.

- **Violation Example:** A `Student` table where the `Subjects` column stores "Math, English, Science".
- **1NF Solution:** Split the multi-valued attribute into separate rows or a new table, so each cell holds a single value.

StudentID	Name	Subject
1	Jaya prakash	Math
1	Jaya prakash	English
2	Arjun	Science

Second Normal Form (2NF)

A table is in 2NF if it is in 1NF and all non-key attributes are **fully functionally dependent on the entire primary key**. This specifically applies to tables with composite keys (keys made of multiple columns) and eliminates partial dependencies.

- **Violation Example:** A table with a composite key `(StudentID, Subject)`, where `Teacher` depends only on `Subject`, not on the full key `(StudentID, Subject)`. This is a partial dependency.

StudentID	Subject	Teacher
1	Math	Mr. A
1	English	Mr. B
2	Math	Mr. A

- **2NF Solution:** Decompose the table into two: one for enrollment (`StudentSubject`) and one for subject details (`SubjectDetails`).

StudentID	Subject
1	Math
1	English
2	Math

Subject	Teacher
Math	Mr. A
English	Mr. B

Third Normal Form (3NF)

A table is in 3NF if it is in 2NF and has **no transitive dependencies**. A transitive dependency occurs when a non-key attribute depends on another non-key attribute.

- **Violation Example:** A `Student` table with primary key `StudentID`, where `DeptName` depends on `DeptID`, and `DeptID` depends on `StudentID`. This is a transitive dependency (`StudentID` → `DeptID` → `DeptName`).

StudentID	Name	DeptID	DeptName
1	Vishnu	101	Science

2	Pradeep	102	Arts

- **3NF Solution:** Decompose into a `Student` table and a `Department` table.

StudentID	Name	DeptID
1	Vishnu	101
2	Pradeep	102

DeptID	DeptName
101	Science
102	Arts

Boyce-Codd Normal Form (BCNF)

BCNF, sometimes called 3.5NF, is a stricter version of 3NF. A table is in BCNF if, for every non-trivial functional dependency X → Y, X must be a **super key** (a set of attributes that uniquely identifies a row).

- **Violation Example:** A table where `{Student, Subject}` determines `Instructor`, but `Instructor` also determines `Subject` (due to a rule that each subject is taught by only one specific instructor). If `{Student, Instructor}` is also a candidate key, a BCNF violation can occur if an instructor teaches multiple subjects.
- **BCNF Solution:** Decompose the table to ensure all determinants are super keys. BCNF typically solves anomalies that 3NF might miss in edge cases with overlapping candidate keys.

Decomposition

The term decomposition refers to the process in which we break down a table in a database into various elements or parts. Thus, decomposition replaces a given relation with a collection of various smaller relations. Thus, in a database, we can make any table break down into multiple tables when we want to collect a particular set of data.

Decomposition must always be lossless. This way, we can rest assured that the data/information that was there in the original relation can be reconstructed accurately on the basis of the

decomposed relations. In case the relation is not decomposed properly, then it may eventually lead to problems such as information loss.

Types of Decomposition

Decomposition is of two major types in DBMS:

- Lossless
- Lossy

1. Lossless Decomposition

A decomposition is said to be lossless when it is feasible to reconstruct the original relation R using joins from the decomposed tables. It is the most preferred choice. This way, the information will not be lost from the relation when we decompose it. A lossless join would eventually result in the original relation that is very similar.

For example,

Let us take ‘A’ as the Relational Schema, having an instance of ‘a’. Consider that it is decomposed into: A1, A2, A3, ... An; with instance: a1, a2, a3, ... an, If a1 ⋈ a2 ⋈ a3 . . . . ⋈ an, then it is known as ‘Lossless Join Decomposition’.

2. Lossy Decomposition

Just like the name suggests, whenever we decompose a relation into multiple relational schemas, then the loss of data/information is unavoidable whenever we try to retrieve the original relation.

Properties of Decomposition

Decomposition must have the following properties:

1. Decomposition Must be Lossless

2. Dependency Preservation

3. Lack of Data Redundancy

1. Decomposition Must be Lossless

Decomposition must always be lossless, which means the information must never get lost from a decomposed relation. This way, we get a guarantee that when joining the relations, the join would eventually lead to the same relation in the result as it was actually decomposed.

2. Dependency Preservation

Dependency is a crucial constraint on a database, and a minimum of one decomposed table must satisfy every dependency. If {P → Q} holds, then two sets happen to be dependent functionally. Thus, it becomes more useful when checking the dependency if both of these are set in the very same relation. This property of decomposition can be done only when we maintain the functional dependency. Added to this, this property allows us to check various updates without having to compute the database structure's natural join.

3. Lack of Data Redundancy

It is also commonly termed as a repetition of data/information. According to this property, decomposition must not suffer from data redundancy. When decomposition is careless, it may cause issues with the overall data in the database. When we perform normalization, we can easily achieve the property of lack of data redundancy.

Transaction concept

A **transaction** in a relational database management system (RDBMS) is a sequence of database operations treated as a single, indivisible logical unit of work. Its purpose is to maintain data integrity and consistency by guaranteeing that either all operations within the unit are completed successfully (committed) or all are canceled (rolled back).

Key Properties: ACID

Every transaction in a reliable database system adheres to the ACID properties:

- **Atomicity:** Guarantees the "all or nothing" principle. The entire transaction either succeeds or fails completely. If any part fails, the whole transaction is undone.

- **Consistency:** Ensures that a transaction moves the database from one valid and consistent state to another, always adhering to predefined rules and constraints.
- **Isolation:** Ensures that multiple transactions running concurrently do not interfere with each other. Each transaction runs as if it is the only one operating on the database at that time.
- **Durability:** Guarantees that once a transaction is successfully committed, its changes are permanent and will survive any future system failures, such as power outages or crashes.

Example: Online Bookstore Order

Consider an online bookstore where a customer orders a book. This single business event involves multiple operations across different tables that must all succeed or fail together to maintain database integrity.

The Operations

Assume the following tables: `BOOKS` (with `Stock` quantity) and `ORDERS`.

The transaction involves these steps in an RDBMS using Transaction Control Language (TCL) commands:

```
sql
-- Start the transaction
BEGIN TRANSACTION;

-- 1. Check book availability and deduct from stock
UPDATE BOOKS
SET Stock = Stock - 1
WHERE BookID = 101 AND Stock >= 1;
-- Check if the update succeeded (e.g., if stock was sufficient)
-- In real applications, checks and error handling are more complex
-- 2. Insert the new order record
INSERT INTO ORDERS (OrderID, CustomerID, BookID, OrderDate)
VALUES (5001, 001, 101, CURRENT_TIMESTAMP);

-- If all operations succeed, commit the changes permanently
COMMIT;
```

Reasoning with ACID Properties

- **Atomicity:** If the `UPDATE` succeeds but the `INSERT` fails (e.g., duplicate OrderID), the entire transaction is rolled back. The book stock is reverted to its original amount, ensuring no stock is lost in a failed order.
- **Consistency:** The database ensures that the `Stock` quantity never drops below zero (an integrity constraint).
- **Isolation:** If two customers try to buy the last copy of the same book simultaneously, the database's concurrency control mechanisms (like locking) ensure that one transaction completes entirely before the other can access the data, preventing both from mistakenly thinking the book is available.
- **Durability:** Once the `COMMIT` statement executes, even if the server immediately crashes, the changes are logged and made permanent when the system recovers.

If an error occurs, the `ROLLBACK` command is used to return the database to its state before the transaction began.

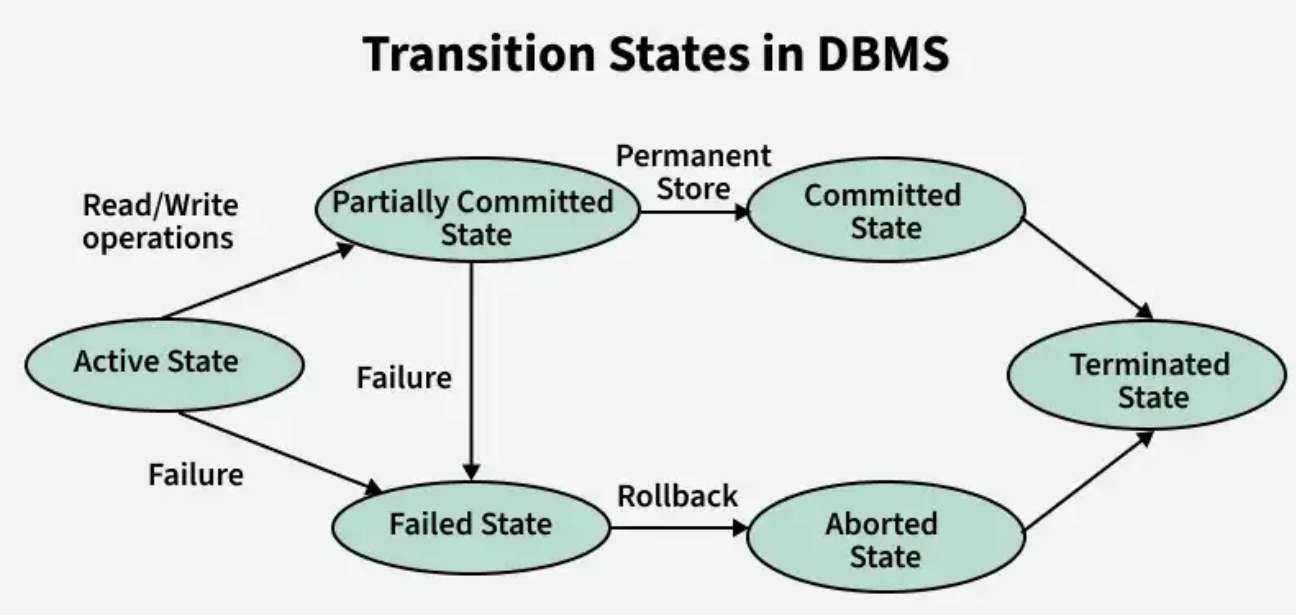
Transaction states

Transaction states represent the different phases a transaction goes through during its lifecycle in a database system. These states help track the progress and outcome of a transaction to ensure data consistency and integrity.

Different Types of Transaction States in DBMS

These are the different types of Transaction States:

1. Active State
2. Partially Committed State
3. Committed State
4. Failed State
5. Aborted State
6. Terminated State



1. Active State

- It is the first stage of any transaction when it has begun to execute. The execution of the transaction takes place in this state.
- Operations such as insertion, deletion, or updation are performed during this state.
- During this state, the data records are under manipulation and they are not saved to the database, rather they remain somewhere in a buffer in the main memory.

2. Partially Committed

- The transaction has finished its final operation, but the changes are still not saved to the database.
- After completing all read and write operations, the modifications are initially stored in main memory or a local buffer. If the changes are made permanent in the database then the state will change to "committed state" and in case of failure it will go to the "failed state".

3. Committed

This state of transaction is achieved when all the transaction-related operations have been executed successfully along with the Commit operation, i.e. data is saved into the database after the required manipulations in this state. This marks the successful completion of a transaction.

4. Failed State

If any of the transaction-related operations cause an error during the active or partially committed state, further execution of the transaction is stopped and it is brought into a failed state. Here, the database recovery system makes sure that the database is in a consistent state.

5. Aborted State

If a transaction reaches the failed state due to a failed check, the database recovery system will attempt to restore it to a consistent state. If recovery is not possible, the transaction is either rolled back or cancelled to ensure the database remains consistent.

In the aborted state, the DBMS recovery system performs one of two actions:

- **Kill the transaction**: The system terminates the transaction to prevent it from affecting other operations.
- **Restart the transaction**: After making necessary adjustments, the system reverts the transaction to an active state and attempts to continue its execution.

6. Terminated State

It refers to the final state of a transaction, indicating that it has completed its execution. Once a transaction reaches this state, it has either been successfully committed or aborted. In this state, no further actions are required from the transaction, as the database is now stable.

Example of Transaction States

Imagine a bank transaction where a user wants to transfer $500 from **Account A** to **Account B**. The system should handle the following transaction states:

Active State

- The transaction begins. It reads the balance of Account A and checks if it has enough funds.
- **Example:** Read balance of Account A = $1000.

Partially Committed State

- The transaction performs all its operations but hasn’t yet saved (committed) the changes to the database.
- **Example:** Deduct $500 from Account A's balance ($1000 - $500 = $500) and temporarily update Account B's balance (add $500).

Committed State

- The transaction successfully completes, and the changes are saved permanently in the database.

- **Example:** Account A's new balance = $500; Account B's new balance = $1500. Changes are written to the database.

Failed State

- If something goes wrong during the transaction (e.g., power failure, system crash), the transaction moves to this state.
- **Example:** System crashes after deducting $500 from Account A but before adding it to Account B.

Aborted State

- The failed transaction is rolled back, and the database is restored to its original state.
- **Example:** Account A's balance is restored to $1000, and no changes are made to Account B.

These states ensure that either the transaction completes successfully (committed) or the database is restored to its original state (aborted), maintaining consistency and preventing errors.

Concurrency execution

Concurrency execution in a Database Management System (DBMS) involves the interleaved or simultaneous operation of multiple transactions to enhance performance, while key concepts and protocols ensure the data remains correct and reliable. This is critical for systems with many simultaneous users, like online banking or e-commerce platforms.

Key Concepts in Concurrency Control

The goal of concurrency control is to maintain the database's integrity despite overlapping operations. This is framed by the **ACID properties**:

- **Atomicity:** Transactions are "all or nothing." If a failure occurs, the entire transaction is undone (rolled back).
- **Consistency:** Each transaction must move the database from one valid state to another.
- **Isolation:** The execution of one transaction must be isolated from others. The result of concurrent execution must be the same as if the transactions ran sequentially (**serializability**).
- **Durability:** Committed changes are permanent and survive system crashes.

Concurrency Execution Example: The "Unrepeatable Read" Problem

The "Unrepeatable Read" scenario occurs when a transaction reads the same data item twice but gets different values because another committed transaction modified the item in between the two reads.

Scenario:

- A user wants to check the current stock count for an item (e.g., 50 units) and then confirm an order that depends on that count.

Execution without Strict Isolation (e.g., in a weak isolation level):

Time	Transaction T1 (Check stock & Order)	Transaction T2 (Stock Restock)	Database State (Stock Count)
T1	Read Stock Count -> 50		50
T2	(User is reviewing the item details)		50
T3		Read Stock Count -> 50	50
T4		Add 20 to Stock Count: 50 + 20 = 70	50
T5		Write Stock Count -> 70	70
T6		**Commit T2**	70
T7	Read Stock Count again -> 70		70
T8	(T1 now sees a different value than the first read)		70

Problem: T1 sees a conflict. The first time it checked the stock, it was 50; the second time, it was 70. This unrepeatable read can lead to logic errors within T1—the initial condition T1 evaluated is no longer true when it attempts to complete its operation.

Concurrency Control Mechanism: Strict Two-Phase Locking (2PL)

To prevent unrepeatable reads, DBMSs implement stricter **isolation levels**, typically using locking mechanisms.

- **Shared Locks (S-locks):** Multiple transactions can hold a shared lock simultaneously (for reading data).
- **Exclusive Locks (X-locks):** Only one transaction can hold an exclusive lock at a time (for writing data).

In the example above, if T1 uses a strict isolation level that prevents unrepeatable reads, T1 would place a **shared lock** on the stock count at T1. When T2 attempts to place an **exclusive lock** to modify the stock at T4, T2 is blocked until T1 completes its entire operation and releases its shared lock, thus guaranteeing that T1 always reads the same consistent value within its duration.

Serializability

Serializability in a Database Management System (DBMS) is a property that ensures concurrent transaction execution produces the same result as some serial execution (transactions running one after another). This maintains data **consistency** and prevents concurrency issues like lost updates or dirty reads.

Types of Serializability

There are two primary types of serializability:

- **Conflict Serializability**: A schedule is conflict-serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations conflict if they belong to different transactions, access the same data item, and at least one is a write operation.
- **View Serializability**: This is a less strict form that focuses only on the final outcome, ensuring the final database state and read operations match those of a serial schedule.

Example: The Bank Account Scenario

Consider a bank account with an initial balance of $1000. Two transactions occur concurrently:

- **T1**: Deposits $100

- **T2**: Withdraws $50

Time	Transaction (T1)	Transaction (T2)	Data Item (Balance)	Explanation
1	Read Balance (1000)		1000	T1 reads the initial balance.
2	Balance = 1000 + 100		1000	T1 calculates the new balance ($1100) locally.
3		Read Balance (1000)	1000	T2 reads the initial balance (a *dirty read* or *lost update* issue if not managed).
4		Balance = 1000 - 50	1000	T2 calculates the new balance ($950) locally.
5	Write Balance (1100)		1100	T1 writes its new balance.
6		Write Balance (950)	950	T2 writes its new balance, overwriting T1's update.

In this non-serializable, interleaved schedule, the final balance is **$950**, which is incorrect (it should be $1000 + $100 - $50 = $1050).

A **serializable schedule** would execute the transactions sequentially, for example, running all of T1 first, then all of T2, ensuring the correct result of $1050:

Time	Transaction (T1)	Transaction (T2)	Data Item (Balance)	Explanation
1	Read Balance (1000)		1000	T1 reads the balance.
2	Balance = 1000 + 100		1000	T1 calculates the new balance.

3	Write Balance (1100)		1100	T1 updates the balance in the database.
4		Read Balance (1100)	1100	T2 reads the updated balance.
5		Balance = 1100 - 50	1100	T2 calculates the new balance ($1050).
6		Write Balance (1050)	1050	T2 updates the balance.

How DBMS Ensures Serializability

DBMS uses concurrency control protocols to ensure serializability. Common techniques include:

- **Lock-Based Protocols**: Using shared (read) and exclusive (write) locks, most notably the Two-Phase Locking (2PL) protocol, to restrict access to data items.
- **Timestamp Ordering**: Assigning unique timestamps to transactions and ordering their execution based on these timestamps.
- **Optimistic Concurrency Control**: Allowing transactions to run freely and checking for conflicts only during the commit phase, rolling back if a conflict is found.
- **Precedence Graph (Serialization Graph)**: A graph used to test if a schedule is conflict-serializable. If the graph has no cycles, the schedule is serializable; otherwise, it is not.

Recoverability

Recoverability in a DBMS is the ability to restore a database to a consistent state after a system failure by undoing uncommitted changes and reapplying committed ones, ensuring data integrity. A common example is a bank transfer, where the system must use **transaction logs** to roll back an uncommitted debit if a crash occurs before the credit is written, or to redo the credit if the debit was already committed but the crash happened before the system could write the credit to the database.

Concepts of recoverability

- **Atomicity:** Guarantees that a transaction is an all-or-nothing operation. If it fails, all its changes are undone, and if it succeeds, all changes are permanent.
- **Consistency:** Ensures that a transaction brings the database from one valid state to another.
- **Durability:** Ensures that once a transaction is committed, its changes are permanent and will survive subsequent system failures.
- **Isolation:** Ensures that concurrent transactions do not interfere with each other.
- **Transaction logs:** A log file on stable storage records every change made to the database. It is crucial for recovery, as it contains the information needed to undo or redo transactions.
- **Checkpointing:** A technique that periodically saves the database state to disk, allowing the system to restart the recovery process more quickly after a crash by not having to re-examine the entire transaction log.
- **Cascading rollback:** A scenario where, if a transaction fails, it forces other dependent transactions to be rolled back as well because they have read its uncommitted data. To prevent this, systems use techniques like cascadeless schedules, where a transaction only reads data that has already been committed.

Example: Bank transfer

1. **Transaction:** A user transfers $100 from account A to account B. This is typically implemented as a single transaction consisting of two operations:
    1. `T1: Write(A) - 100` (Debit $100 from account A)
    2. `T2: Write(B) + 100` (Credit $100 to account B)
2. **Logging:** Before executing `Write(A)`, the system writes a log entry like `[A, old_value, new_value]` (e.g., `[A, 500, 400]`) to stable storage. It does the same for account B, e.g., `[B, 300, 400]`.
3. **Crash Scenario:** Suppose the system crashes after `T1` is written to disk but before `T2` is written, and a crash occurs before the commit is officially logged.
4. **Recovery:**
    1. The DBMS starts its recovery process and reads the logs.
    2. It finds that transaction `T1` was committed (its log entries and a "commit" record exist), but `T2` was not.
    3. To restore consistency, the system uses the log entries to **undo** the changes of `T1` by restoring account A to its original value (e.g., $500). This ensures atomicity.
5. **Another Crash Scenario:** If the crash occurred after both `T1` and `T2` were logged but before the commit was officially logged for both.

6. **Recovery:**
    1. The system reads the logs and finds that `T1` and `T2` are both present in the log with "commit" records.
    2. To restore durability, the system uses the log entries to **redo** both `T1` and `T2`, ensuring that account A is debited by $100 and account B is credited by $100

The key is that the system can use the log to either undo incomplete operations or redo completed ones, which allows it to guarantee that the database is brought to a consistent and correct state after any failure.

Testing for Serializability

Serializability is a property of a concurrent transaction schedule that ensures the final database state is the same as if the transactions were executed one after another in some serial order. Testing for serializability typically involves using a **precedence graph** (or serialization graph) to check for cycles.

Testing for Conflict Serializability

Conflict serializability is the most common and practical form of serializability testing. Two operations conflict if they belong to different transactions, access the same data item, and at least one of them is a write operation (e.g., read-write, write-read, or write-write conflicts).

The testing method uses a precedence graph:

Steps to Construct and Test a Precedence Graph:

1. **Create Nodes**: For every transaction (T_i) in the schedule, draw a node in the graph.
2. **Draw Directed Edges**: For each pair of conflicting operations, draw a directed edge from the transaction that executed first to the transaction that executed second.
    - If T_i writes to data item X before T_j reads X (write-read conflict), draw an edge from T_i to T_j ($T_i \rightarrow T_j$).
    - If T_i reads X before T_j writes to X (read-write conflict), draw an edge from T_i to T_j ($T_i \rightarrow T_j$).
    - If T_i writes to X before T_j writes to X (write-write conflict), draw an edge from T_i to T_j ($T_i \rightarrow T_j$).

Check for Cycles: Examine the resulting graph for any cycles (loops).

- **If there is no cycle**, the schedule is **conflict serializable** and thus consistent.
- **If a cycle exists**, the schedule is **not conflict serializable**, meaning it cannot be rearranged into an equivalent serial order and may lead to data inconsistency

Example:

Consider the following schedule S with two transactions, T_1 and T_2, operating on data item A:

Time	T_1	T_2
1	R(A)	
2		W(A)
3	W(A)	
4		R(A)

Analysis:

- T_1 reads A (at time 1) before T_2 writes A (at time 2) -> **Conflict** ($T_1 \rightarrow T_2$).
- T_2 writes A (at time 2) before T_1 writes A (at time 3) -> **Conflict** ($T_2 \rightarrow T_1$).
- T_1 writes A (at time 3) before T_2 reads A (at time 4) -> **Conflict** ($T_1 \rightarrow T_2$).
- T_2 writes A (at time 2) before T_2 reads A (at time 4) - this is within the same transaction and not a conflict.

Precedence Graph:

The conflicts create edges $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$. This forms a cycle: $T_1 \leftrightarrow T_2$.

Conclusion: The schedule is **not conflict serializable**.

View Serializability

View serializability is a less strict, more general form of serializability, focused on whether a non-serial schedule produces the same *final output* as some serial schedule. It is computationally harder to test than conflict serializability and involves checking three conditions (initial reads, updated reads, and final writes) for view equivalence to an existing serial schedule. Because all conflict serializable schedules are also view serializable, but not vice-versa, commercial DBMS typically ensure conflict serializability through mechanisms like two-phase locking.