UNIT – 4

1. Redundancy:

Redundancy refers to the practice of storing the same data multiple times within the database or across different parts of the database. This can involve duplicating data in one or more tables or records. Redundancy in a DBMS is generally considered undesirable because it can lead to several problems, such as data inconsistencies, increased storage requirements, and difficulties in maintaining data integrity.

Problems caused by redundancy:

Redundancy in a database management system (DBMS) can lead to several problems, including:

- **Data Inconsistency:** Redundant data can lead to data inconsistencies when the same information is stored in multiple places, and some copies are updated while others are not. This can result in incorrect or conflicting data, making it challenging to maintain data integrity.
- **Increased Storage Costs:** Redundant data consumes additional storage space, which can increase hardware and operational costs. As data redundancy grows, it may require more disk space and memory to store and manage the data.
- **Data Update Anomalies:** Redundancy can lead to data update anomalies, where changes to one instance of data are not propagated consistently to all instances. This can result in data inconsistencies and errors.
- **Complex Maintenance:** Managing redundant data is more complex and time-consuming. When updates or changes are

required, they must be made in multiple places, increasing the likelihood of errors and inconsistencies.

- **Performance Issues**: Redundancy can negatively impact database performance. Querying and updating redundant data takes more time and resources, as the system must manage multiple copies of the same information.
- **Data Integrity Issues:** Redundancy can make it challenging to maintain data integrity and enforce data constraints. For example, if a unique constraint is supposed to apply to a particular attribute, redundancy may make it difficult to ensure uniqueness.
- **Difficulty in Data Migration:** When migrating data from one system to another or consolidating databases, redundant data can lead to complications and increased efforts in data mapping and transformation.
- **Increased Risk of Data Errors**: With redundant data, there's a higher likelihood of data errors, such as typos or inaccuracies, as these errors need to be corrected in multiple places.
- **Inefficient Backup and Recovery:** Backing up and recovering redundant data can be less efficient, as it requires more storage and increases the time needed for these operations.
- **Security Risks:** Redundant data can pose security risks, as sensitive information may be duplicated in multiple locations, increasing the potential for unauthorized access or data breaches.

To mitigate these problems, database designers aim to minimize redundancy through normalization techniques, which involve structuring the database in a way that reduces or eliminates the need for redundant data storage. By doing so, they can improve data

consistency, reduce storage costs, and enhance overall database performance and integrity.

2. Decomposition:

Decomposition in the context of Database Management Systems (DBMS) refers to the process of breaking down a single, complex relation (table) into multiple smaller relations. This process is part of the database normalization technique, which aims to eliminate or reduce data redundancy, maintain data integrity, and improve database efficiency.

The primary goal of decomposition is to structure the database in such a way that each relation (table) contains non-redundant, atomic, and related data. To achieve this, the process involves dividing a large relation into multiple smaller ones, each with its own attributes (columns). This often results in a set of related tables that are connected through primary and foreign keys, which represent the relationships between entities in the database.

Problem caused by decomposition:

Decomposition in a Database Management System (DBMS) is generally aimed at improving the structure and efficiency of a database by eliminating data redundancy and ensuring data integrity. However, it's essential to note that decomposition, when taken to extreme levels or applied inappropriately, can sometimes lead to certain problems:

- Storage Overhead: In some cases, decomposition can lead to an increase in the number of tables and indexes, which can consume additional storage space. While storage is often

inexpensive, excessive storage overhead can still be a concern for large databases.

- Data Retrieval Challenges: Retrieving data from highly normalized tables may require the use of numerous JOIN operations, making it more difficult for developers to retrieve and analyse data.
- Data Integrity Issues: In some cases, decomposition can result in anomalies related to data insertion, updates, and deletions. For instance, it may become challenging to ensure that the database maintains referential integrity when there are many interrelated tables.
- Application Compatibility: If the decomposition is not well-aligned with the requirements of the applications using the database, it can lead to compatibility issues and additional coding complexity.

To avoid these problems, it's important to strike a balance between normalization and performance. Database designers need to consider the specific requirements of their applications and choose an appropriate level of decomposition that optimizes query performance while still maintaining data integrity. In practice, many databases aim for third normal form (3NF) or Boyce-Codd Normal Form (BCNF) as a good compromise between normalization and performance.

3. Functional Dependencies:

A functional dependency is a relationship between two sets of attributes in a relation (table) such that for each combination of values in one set, there is a unique combination of values in the

other set. In other words, it specifies how knowing the value of one attribute(s) uniquely determines the value of another attribute(s).

Reasoning about functional dependencies is a crucial aspect of database design and normalization in Database Management Systems (DBMS). Functional dependencies are constraints that define how attributes (columns) in a database table are related to each other. They play a fundamental role in ensuring data integrity and minimizing redundancy. Here's why reasoning about functional dependencies is important:

Reasoning about Functional dependencies:

- Data Integrity: Functional dependencies help ensure the accuracy and consistency of data in a database. By specifying how attributes are related, they prevent the insertion of inconsistent or erroneous data.
- Normalization: Functional dependencies are used to identify and eliminate redundancy in a database. This is done through a process known as normalization, which aims to organize data efficiently by breaking it down into smaller tables. The identification of functional dependencies is a key step in this process.
- Retrieval Optimization: Understanding the functional dependencies in a database allows for more efficient query optimization. The query optimizer can use this information to select the most efficient execution plan, reducing the time and resources required to retrieve data.
- Data Modification Anomalies: Functional dependencies help in identifying and addressing data modification anomalies, such as insertion, update, and deletion anomalies. By specifying how

data is related, you can design the database structure to minimize these anomalies.

- Data Redundancy Reduction: Understanding the functional dependencies helps in creating well-structured and normalized databases, reducing data redundancy. Redundant data is not only inefficient but can also lead to inconsistencies when data is updated.

4. Normalization:

Normalization is a database design process that helps organize data in a structured and efficient way, aiming to reduce redundancy and maintain data integrity. It involves breaking down large, complex tables into smaller, related tables and establishing relationships between them. The primary goal of normalization is to minimize data anomalies (as discussed earlier) while maintaining data consistency and integrity. Normalization is usually done through a series of steps, each referred to as a normal form. There are several normal forms, including First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and more.

Types:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

i.) First Normal Form (1NF)

First Normal Form (1NF) is the first step in the normalization process of a relational database. It ensures that the data is organized in a way that there is no redundancy within each row of a table. In 1NF:

- Each column in a table must contain only atomic (indivisible) values.
- Each row must be unique, and there must be a way to distinguish each row from every other row (usually by using a primary key).

Here's an explanation of 1NF with an example table:

Consider a table that contains information about books and their authors. Each row in the table represents a book, and it has the following columns: BookID, Title, Authors (a list of authors separated by commas), and Publication Year.

BookID	Title	Authors	PublicationYear
1	"Book1" “Book2”	"Author1”	2020
2	"Book3"	"Author2"	2019
3	"Book4" “Book5”	"Author3"	2021

This table is not in 1NF for the following reasons:

- The "Authors" column contains a list of authors separated by commas. This is not atomic; it violates the 1NF rule. In 1NF, each column should have atomic values.
- The "Authors" column contains multiple values. In a relational database, we should aim for a single value in each cell, and if there are multiple values, they should be placed in separate rows or in a related table.

To bring this table into 1NF, we need to split the Authors information into a separate table and establish relationships between the two tables.

Books Table (1NF):

BookID	Title	Publication Year
1	"Book1"	2020
2	"Book2"	2020
3	"Book3"	2019
4	“Book4”	2021
5	“Book5”	2021

Authors Table (1NF):

AuthorID	Author
1	"Author1"
2	"Author2"
3	"Author3"

Books & Author table:

BookId	AuthorId
1	1
2	1
3	2
4	3
5	3

Now, the data is in 1NF. The "Authors" information has been split into a separate table, and the Books and Authors tables are related through a common key (AuthorID in Authors and BookID in Books). This eliminates the issues of non-atomic values and the presence of multiple values in a single cell, making the tables compliant with 1NF.

ii.) Second Normal Form:

Second Normal Form (2NF) is the next step in the normalization process of a relational database. It builds upon the requirements of First Normal Form (1NF) and focuses on eliminating partial dependencies in a table. In 2NF:

- The table must already be in 1NF.
- All non-key attributes (attributes not part of the primary key) must be fully functionally dependent on the entire primary key. This means that non-key attributes should depend on the whole primary key, not just part of it.

To understand 2NF better, let's examine an example table and then transform it into 2NF:

StudentCourses Table (not in 2NF):

StudentID	StudentName	CourseID	CourseName	Instructor
101	Alice	201	Math 101	Mr. Smith
101	Alice	202	Science 101	Mrs. John
102	Bob	201	Math 101	Mr. Smith
103	Carol	203	History 101	Mr. Davis

In this table, StudentID and CourseID together can be used as the primary key since each combination is unique. This table is in 1NF

because it contains only atomic values. However, it's not in 2NF because there's a partial dependency. StudentID and CourseID have a duplicate values.

To bring this table into 2NF, we create two separate tables: one for student information and another for course enrollments.

Students Table (2NF):

StudentID	StudentName
101	Alice
102	Bob
103	Carol

Courses Table (2NF):

CourseID	CourseName	Instructor
201	Math 101	Mr. Smith
202	Science 101	Mrs. Johnson
203	History 101	Mr. Davis

Enrollments Table (2NF):

StudentID	CourseID
101	201
101	202
102	201

103	203

Now, the data is in 2NF. The partial dependency issue has been resolved by creating separate tables for students and courses, and the information in the Enrollments table depends on the entire primary key, which is the combination of StudentID and CourseID. This satisfies the requirements of 2NF.

iii.) Third Normal Form:

Third Normal Form (3NF) is the next step in the normalization process of a relational database. It builds upon the requirements of First Normal Form (1NF) and Second Normal Form (2NF) and focuses on eliminating transitive dependencies within a table. In 3NF:

- The table must already be in 2NF.
- There should be no transitive functional dependencies, meaning that non-key attributes should not depend on other non-key attributes.

Let's illustrate the concept of 3NF with an example table:

Consider a table that tracks student enrollments in courses with fees structure:

Name	Roll No	Branch	Fees
Dilip	1	Csc	95000
Smith	2	DS	250000
John	3	AIML	200000
kuhan	4	ECE	115000

In this table, Roll No is used as the primary key since each combination is unique. This table is in 2NF because it doesn't have partial dependencies. However, it's not in 3NF because there is a transitive dependency. The Fees column depends upon the branch column but not directly on primary column Roll No.

To bring this table into 3NF, we need to create separate tables for student information, course information, and instructor information, eliminating the transitive dependency:

Students Table (3NF):

Name	Roll No
Dilip	1
Smith	2
John	3
kuhan	4

Courses Table (3NF):

Branch ID	Branch	Fees
101	Csc	95000
222	DS	250000
312	AIML	200000
408	ECE	115000

Student Enrollments:

Roll No	Branch ID
1	101
2	222

3	312
4	408

Now, the data is in 3NF. We have eliminated the transitive dependency by creating separate tables for students and courses and the data is organized in a more normalized structure that meets the requirements of 3NF.

iv.) Boyce Codd Normal Form (3.5NF):

Boyce-Codd Normal Form (BCNF) is a higher level of database normalization that refines the concepts of First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF). BCNF is designed to address certain types of redundancy and anomalies that can still occur in 3NF tables. In BCNF:

- The table must be in 1NF, 2NF, and 3NF.
- It must satisfy an additional constraint: for every non-trivial functional dependency (X → Y), X must be a superkey. A superkey is a set of attributes that can uniquely identify a row in a table.

To illustrate BCNF, let's consider an example table:

Suppose we have a table that tracks students and the courses they have registered for, along with the course instructors. The table has

the following columns: StudentID, StudentName, CourseID, CourseName, and InstructorName.

StudentID	StudentName	CourseID	CourseName	InstructorName
101	Alice	201	Math 101	Mr. Smith
102	Bob	202	Science 101	Mrs. Johnson
103	Carol	201	Math 101	Mr. Smith
104	Dave	203	History 101	Mr. Davis

In this table, the primary key could be the combination of StudentID and CourseID because each combination is unique. This table is in 3NF since it doesn't have transitive dependencies, but it's not in BCNF because not every functional dependency satisfies the requirement of X being a superkey.

In this table, consider the functional dependency: CourseName → InstructorName. CourseName is not a superkey because it doesn't uniquely identify a student's registration, yet InstructorName depends on CourseName. This is a violation of BCNF.

To bring this table into BCNF, we need to decompose it into separate tables to eliminate the violation:

Courses Table (BCNF):

CourseID	CourseName
201	Math 101
202	Science 101
203	History 101

Instructors Table (BCNF):

CourseName	InstructorName
Math 101	Mr. Smith
Science 101	Mrs. Johnson
History 101	Mr. Davis

Students Table (BCNF):

StudentID	StudentName
101	Alice
102	Bob
103	Carol
104	Dave

Enrollments Table (BCNF):

StudentID	CourseID
101	201
102	202
103	201
104	203

Now, the data is in BCNF. The functional dependency issues have been resolved, and every non-trivial dependency satisfies the requirement of X being a superkey. The data is organized in a more normalized structure that adheres to BCNF.

v.) Fourth Normal Form:

Fourth Normal Form (4NF) is a level of database normalization that refines the concepts of First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF). 4NF is designed to address multi-valued dependencies, which occur when an attribute depends on a combination of attributes rather than a single attribute. In 4NF:

- The table must be in 1NF, 2NF, 3NF, and BCNF.
- It must not have any multi-valued dependencies, which means that an attribute should not depend on a combination of attributes that act as a composite key.

To illustrate 4NF, let's consider an example table:

StudentID	StudentName	Courses	Hobby
101	Alice	CS	Cricket
102	Bob	DS	Chess
103	Cat	AIML	Volleyball
104	Dog	ECE	Football
105	Meow	Civil	Cricket

In this table, the primary key is StudentID. This table is in 3NF and BCNF, but it's not in 4NF due to multi-valued dependencies.

The multi-valued dependency arises because StudentName, Courses, Hobby depends on primary key column StudentId. This can result in data redundancy and inconsistencies.

To bring this table into 4NF, we need to decompose it into separate tables to eliminate multi-valued dependencies:

Students Table (4NF):

StudentID	StudentName
101	Alice
102	Bob
103	Cat
104	Dog
105	Meow

Courses Table (4NF):

StudentID	Courses
101	CS
102	DS
103	AIML
104	ECE
105	Civil

Hobbies Table (4NF):

StudentID	Hobby
101	Cricket
102	Chess
103	Volleyball
104	Football
105	Cricket

Now, the data is in 4NF. The multi-valued dependency issues have been resolved, and each attribute depends only on the primary key, eliminating redundancy and adhering to the requirements of 4NF.

vi.) Fifth Normal Form:

Fifth Normal Form (5NF) is the highest level of database normalization and addresses join dependencies in a relational database. Join dependencies occur when a database schema is such that certain facts can only be represented by joining multiple tables. 5NF is rarely used in practice and is typically applied in very complex database scenarios, such as in data warehousing and some specialized research databases.

To illustrate 5NF, let's consider an example table:

Suppose we have a database for a university that stores information about courses, students, and their enrollments. The table has the following columns: CourseID, CourseName, StudentID, StudentName, and Grade.

Subject	Faculty	Year
Data Structure	Sivan Raj	2
Data Structure	Agastin	2
DBMS	Vignesha	2
Java	Jamil	1
AJP	Balaji	3

In this table is in 3NF, BCNF, and 4NF but not in 5NF because it exhibits a join dependency: to determine a student's grade in a course, you must join two or more tables.

To bring this data into 5NF, you would need to create separate tables:

Subject Table:

Subject	Faculty
Data Structure	Sivan Raj
Data Structure	Agastin
DBMS	Vignesha
Java	Jamil
AJP	Balaji

Faculty Table:

Faculty	Year
Sivan Raj	2
Agastin	2
Vignesha	2
Jamil	1
Balaji	3

Students Table:

Subject	Year
Data Structure	2
Data Structure	2
DBMS	2
Java	1
AJP	3

Join Dependencies:

To represent the join dependencies explicitly, you may introduce a way to define how these tables should be joined, such as:

- A view that combines the relevant tables based on defined join conditions.
- Foreign keys that link the tables together.

In 5NF, the goal is to eliminate redundancy and ensure that each fact is stored in the smallest number of tables possible. This level of normalization is not commonly used in most practical database design scenarios but becomes relevant in specialized situations where minimizing redundancy and join complexity is crucial.

Transaction:

In a Database Management System (DBMS), a transaction is a fundamental concept that ensures the integrity and consistency of a database. Transactions are a way to group one or more database operations into a single unit of work that is either fully completed or fully aborted, without leaving the database in an inconsistent state. Here's a detailed explanation of the transaction concept in DBMS:

What is a Transaction?

A transaction is a sequence of one or more SQL statements that are treated as a single, indivisible unit of work.

Transactions in DBMS follow the ACID properties, which are key characteristics ensuring data reliability:

- **Atomicity:** A transaction is atomic, meaning that it is an all-or-nothing operation. Either all the changes made by the transaction are applied, or none of them are.
- **Consistency:** A transaction brings the database from one consistent state to another consistent state. It should satisfy integrity constraints.
- **Isolation:** Transactions are executed in isolation from each other, ensuring that the operations of one transaction are not visible to others until it's completed.
- **Durability:** Once a transaction is committed, its changes are permanent and will survive any system failures.

Atomicity:

- Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within a transaction are completed successfully, or none of them are.

Example:

Imagine a banking application where you need to transfer money from one account to another. If a power outage occurs or any part of the transaction fails, you don't want the money to be lost or left in an inconsistent state. Atomicity ensures that the entire transaction is all-or-nothing.

Consistency:

- Consistency ensures that a transaction brings the database from one consistent state to another consistent state. It enforces integrity constraints on the data.

Example:

In a student registration system, suppose a constraint requires that a student cannot be registered for more than 20 credit hours. If a registration transaction attempts to register a student for 25 credit hours, the system should detect this violation and prevent the transaction from proceeding.

Isolation:

- Isolation ensures that concurrent execution of transactions does not result in interference or data inconsistency. One

transaction's operations are isolated from the operations of other transactions until they are committed.

Example:

Consider two users simultaneously trying to book the last seat on a flight. Isolation prevents one user from booking the seat while the other is in the process of booking. Both transactions will complete successfully without interfering with each other.

Durability:

- Durability ensures that once a transaction is committed, its changes are permanent and will survive system failures such as crashes or power outages.

Example:

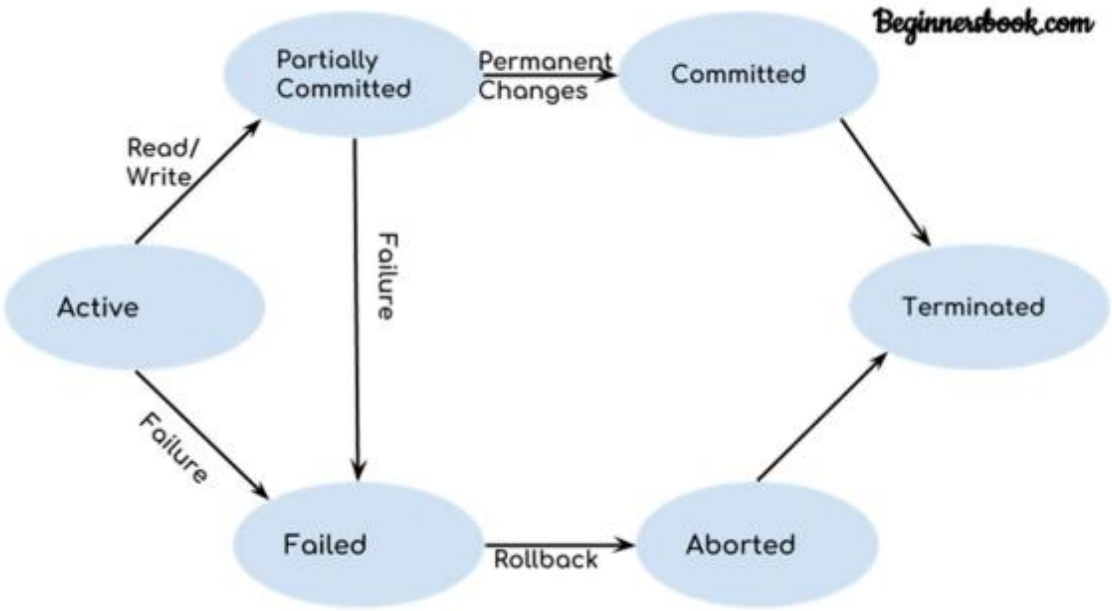
When you withdraw money from an ATM and receive a printed receipt, you expect the transaction to be durable. Even if the ATM crashes immediately after the transaction, your account balance should reflect the withdrawal when the system comes back online.

Transaction States:

Transactions in DBMS typically go through several states:

- Active: The initial state where the transaction is executing SQL statements.
- Partially Committed: The transaction has executed all its statements, and the DBMS is about to make the changes permanent.

- Committed: The transaction is successfully completed, and its changes are now permanent.
- Failed: If a transaction encounters an error or cannot be completed for any reason, it is marked as failed and must be rolled back.
- Aborted/Rolled Back: In the case of failure, the DBMS ensures that all the changes made by the transaction are undone to maintain consistency.



Lets discuss these states one by one.

Active State

As we have discussed in the DBMS transaction introduction that a transaction is a sequence of operations. If a transaction is in execution then it is said to be in active state. It doesn't matter which step is in execution, until unless the transaction is executing, it remains in active state.

Failed State

If a transaction is executing and a failure occurs, either a hardware failure or a software failure then the transaction goes into failed state from the active state.

Partially Committed

State As we can see in the above diagram that a transaction goes into “partially committed” state from the active state when there are read and write operations present in the transaction. A transaction contains number of read and write operations. Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database.

The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure. This state helps us to rollback the changes made to the database in case of a failure during execution.

Committed State

If a transaction completes the execution successfully then all the changes made in the local memory during partially committed state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

Aborted State

As we have seen above, if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent

state and the transaction goes into aborted state from the failed state.

Transaction Control Statements:

DBMS provides commands to control transactions:

- **BEGIN TRANSACTION** or **START TRANSACTION**: Initiates a new transaction.
- **COMMIT**: Marks a transaction as successful and makes its changes permanent.
- **ROLLBACK**: Undoes all the changes made by the transaction and restores the database to its state before the transaction started.

Let's illustrate the concept of a transaction in a DBMS with a simple example involving a banking system. In this example, we'll cover a money transfer transaction between two bank accounts.

Suppose we have two bank accounts:

Account A with a balance of $1,000.

Account B with a balance of $500.

We want to transfer $200 from Account A to Account B. Here's how the transaction works:

Begin Transaction: We start a transaction with the BEGIN TRANSACTION command.

Transfer Money:

We deduct $200 from Account A.

We add $200 to Account B.

These SQL statements might look like:

UPDATE Account SET Balance = Balance - 200 WHERE AccountNumber = 'A';

UPDATE Account SET Balance = Balance + 200 WHERE AccountNumber = 'B';

Commit Transaction:

If all operations are successful and the funds have been successfully transferred, we use the COMMIT command to make the changes permanent.

End Transaction: The transaction is successfully completed.

Now, let's consider what happens if an error occurs during the transaction:

Begin Transaction: Start the transaction.

Transfer Money:

Deduct $200 from Account A.

An error occurs, preventing the addition of $200 to Account B.

SQL statements:

UPDATE Account SET Balance = Balance - 200 WHERE AccountNumber = 'A';

-- An error occurs here, and the transaction cannot proceed further.

Rollback Transaction:

Since there was an error during the transaction, we use the ROLLBACK command to undo all changes made within this transaction.

End Transaction: The transaction is rolled back, and the database returns to its previous state.