Question 1: Normalization

1NF:

Answer : Our table is already in 1NF since it satisfies the criteria mentioned below.
=>There are no duplicate rows,
each cell contains a single value,
and entries in each column are of the same kind.

2NF:

All other attributes are functionally dependent on the "Book ID". However, attributes like Author, Genre, and Publisher might have their own attributes in a larger database system (for example, Publisher might have an address). But given the current table structure, there's no partial dependency, so the table is already in 2NF.

Third Normal Form (3NF)

In 3NF, we ensure that non-key columns are dependent only on the primary key and not on other non-key columns.

If we observe, there's a potential transitive dependency between `Publisher` and `ISBN` - different books with the same publisher will have different ISBNs, but every unique ISBN will correspond to a single publisher.

We can create separate tables to eliminate this transitive dependency:

Books Table

Book ID	Title	Author	Genre	ISBN	Price
101	To Kill a Mockingbird	Harper Lee	Fiction	978-0061120084	10.99
102	The Great Gatsby	F. Scott Fitzgerald	Fiction	978-0743273565	12.50
103	Principles of Physics	Jearl Walker	Science	978-0321976444	50.00

Publisher Table

ISBN	Publisher
978-0061120084	HarperCollins
978-0743273565	Scribner
978-0321976444	Wiley

This ensures that the database is now in 3NF because there's no non-key column dependent on another non-key column.

[Bonus]Question 2: Database Normalization Practice

For 1NF:

Our table already meets the 1NF requirements. Each column contains atomic values and there aren't repeating groups.

For 2NF:

- Eliminate partial dependencies on a composite key.

The primary key could be a combination of "Employee ID" and "Project ID" since an employee can work on multiple projects and each project can have multiple employees.

However, columns like "Employee Name", "Department", "Salary" are only dependent on the "Employee ID" and columns like "Project Name", "Start Date", "End Date" are only dependent on the "Project ID". This indicates partial dependencies.

Tables after 2NF:

1. Employee Table
    - Employee ID
    - Employee Name
    - Department
    - Salary

2. **Project Table**
    - Project ID
    - Project Name
    - Start Date
    - End Date
3. **EmployeeProject Table** (Associative table to capture many-to-many relationship)
    - Employee ID (Foreign key)
    - Project ID (Foreign key)

For 3NF:

- Remove columns that are not dependent on the primary key.

Upon closer inspection, the tables derived from 2NF are already in 3NF. No column is dependent on another column; they are all dependent on the primary key of their respective tables.

For 4NF:

- Remove multi-valued dependencies.

Considering the "EmployeeProject Table", there's no multi-valued dependency. Each combination of "Employee ID" and "Project ID" is unique, and there aren't attributes that depend on a part of the composite primary key. Therefore, the tables are already in 4NF.

For 5NF:

- Ensure there's no spurious data when decomposed tables are joined back.

Given the current structure, there's no indication of any spurious data when we perform a natural join on the decomposed tables. Thus, the current structure satisfies the requirements for 5NF.

Final Tables:

1. **Employee Table**
    - Employee ID

    - Employee Name
    - Department
    - Salary
2. **Project Table**
    - Project ID
    - Project Name
    - Start Date
    - End Date
3. **EmployeeProject Table**
    - Employee ID (Foreign key)
    - Project ID (Foreign key)

The database is now in 5NF, which ensures minimized redundancy and enhanced data integrity.

Question 3: What are the primary keys and foreign keys in a relational database, and how do they establish relationships between tables?

Primary Keys: A primary key is a unique identifier for a record in a table. It ensures that each record within a table can be uniquely identified by its primary key value. No two records can have the same primary key value, and a primary key column cannot have NULL values. A table can have only one primary key, which may consist of single or multiple columns (composite key).

A foreign key in a table is a reference to the primary key of another table. It establishes a relationship between two tables, allowing for the concept of referential integrity in the database. The table that contains the foreign key is called the referencing or child table, and the table being referred to by the foreign key is called the referenced or parent table. A foreign key ensures that the relationship between tables maintains consistent and valid data.

Establishing Relationships between Tables: Primary keys and foreign keys play a pivotal role in establishing relationships between tables in a relational database.

These relationships can be of the following types:

1. One-to-One Relationship: Each record in Table A relates to one, and only one, record in Table B, and vice versa.
2. One-to-Many Relationship: A single record in Table A can relate to one or more records in Table B, but a record in Table B relates to only one record in Table A.
3. Many-to-Many Relationship: Records in Table A can relate to multiple records in Table B, and records in Table B can relate to multiple records in Table A. This relationship is implemented using a junction table (also known as a bridge or join table) with foreign keys referencing the primary keys of the two tables.

In essence, primary keys ensure each record's uniqueness in a table, while foreign keys establish a linkage and enforce referential integrity between tables. Together, they form the backbone of relationships in relational databases, ensuring data accuracy, consistency, and structural integrity.

Question 4: Explain the ACID properties in the context of database transactions.

Certainly! The ACID properties refer to a set of properties that ensure reliable processing of database transactions. They ensure that when data is committed to a database, it's done safely, even in the face of errors or system failures. The acronym "ACID" stands for Atomicity, Consistency, Isolation, and Durability.

Let's delve into each of these properties:

1. **Atomicity:**
   - Atomicity guarantees that each transaction is treated as a single "unit", which either completes in its entirety or fails completely.
   - This means if a transaction is composed of multiple individual tasks, all tasks must be completed successfully for the transaction to be considered successful. If any task fails, the entire transaction fails, and any changes are rolled back to their previous state.
   - In simple terms, a transaction can't be in a "half-completed" state. It's either fully executed or fully rolled back.
2. **Consistency:**

    - Consistency ensures that a transaction brings the database from one valid state to another valid state.
    - Before a transaction starts, the database is in a consistent state, and after its completion (whether it succeeds or fails), the database must be in another consistent state.
    - Any transaction should preserve the integrity constraints of the database, such as unique, primary, and foreign key constraints.
3. **Isolation:**
    - Isolation ensures that transactions are executed independently and in isolation from one another. This means the intermediate state of a transaction is invisible to other transactions.
    - As a result, if multiple transactions are executed concurrently, the outcome is as if they were executed serially (one after the other). This avoids potential conflicts and data inconsistencies.
    - Database management systems use various locking mechanisms and concurrency control methods to achieve isolation.
4. **Durability:**
    - Durability guarantees that once a transaction is committed, its changes to the database become permanent and will survive any subsequent failures such as system crashes or power losses.
    - This is typically achieved by storing transaction logs or using backup mechanisms. Even if the system encounters a failure, the database can be restored to its last known consistent state using these logs or backups.

In the context of database transactions, the ACID properties play a crucial role in ensuring the reliability and robustness of a database system. They prevent data corruption and ensure that even in the event of system failures, the data remains accurate and reliable.

Question 5: Describe the concept of indexing in a database. How does indexing improve query performance?

Concept of Indexing in a Database:

Indexing in a database is akin to the index in a book. Just as an index in a book helps you find specific content without reading the entire book, a database index allows the database management system (DBMS) to retrieve data without scanning the entire table.

An index provides a structured way to store a subset of data in a table that makes it more efficient to retrieve specific rows. Typically, the index is created on one or more columns of a table. The DBMS maintains the index and ensures it stays up-to-date as data is added, updated, or removed from the table.

How Indexing Improves Query Performance:

1. **Faster Data Retrieval:** Without an index, the DBMS would have to perform a full table scan, reading every row in the table to find the ones that match the query criteria. With an index, the DBMS can directly jump to the desired rows, significantly reducing the number of rows it has to read.
2. **Reduces I/O Operations:** Since the database can quickly locate the rows for a particular query using the index, it significantly reduces the number of disk I/O operations, leading to faster query performance.
3. **Efficient WHERE Clause Execution:** Indexes are especially beneficial for queries that use the WHERE clause. An index on columns used in the WHERE clause can speed up the search for matching rows.
4. **Optimized JOIN Operations:** If tables are joined on columns that are indexed, the join operation can be significantly faster, as the database can quickly identify matching rows.
5. **Sorted Results:** Some indexes, like the B-tree index (commonly used), store data in sorted order. This makes operations like ORDER BY on indexed columns more efficient.
6. **Maintains Data Integrity:** Unique indexes ensure data integrity by preventing duplicate entries in the column(s) on which the index is defined.

However, it's important to note a few things about indexes:

- **Overhead:** While indexes speed up data retrieval, they add some overhead. Creating and maintaining indexes consumes additional disk space. Moreover, every time data is added or updated, the indexes also need to be updated. This can slow down write operations.

- Selectivity: Indexes are most effective when they are highly selective, meaning they help narrow down the results to a small subset of rows. An index on a column with many repeated values (low selectivity) might not be as effective.
- Choosing Indexes: It's essential to choose the right columns to index based on the queries the database will handle. Indexing every column can be counterproductive due to the overhead it introduces.

In summary, indexing is a powerful tool in the realm of databases, aiding in efficiently retrieving data. Properly designed indexes, tailored to fit the specific query patterns of a database, can vastly improve query performance.

Question 6: Explain the concept of concurrency control, deadlocks in a multi-user database environment.

Concept of Indexing in a Database:

Indexing in a database is akin to the index in a book. Just as an index in a book helps you find specific content without reading the entire book, a database index allows the database management system (DBMS) to retrieve data without scanning the entire table.

An index provides a structured way to store a subset of data in a table that makes it more efficient to retrieve specific rows. Typically, the index is created on one or more columns of a table. The DBMS maintains the index and ensures it stays up-to-date as data is added, updated, or removed from the table.

How Indexing Improves Query Performance:

1. Faster Data Retrieval: Without an index, the DBMS would have to perform a full table scan, reading every row in the table to find the ones that match the query criteria. With an index, the DBMS can directly jump to the desired rows, significantly reducing the number of rows it has to read.
2. Reduces I/O Operations: Since the database can quickly locate the rows for a particular query using the index, it significantly reduces the number of disk I/O operations, leading to faster query performance.

3. **Efficient WHERE Clause Execution:** Indexes are especially beneficial for queries that use the WHERE clause. An index on columns used in the WHERE clause can speed up the search for matching rows.
4. **Optimized JOIN Operations:** If tables are joined on columns that are indexed, the join operation can be significantly faster, as the database can quickly identify matching rows.
5. **Sorted Results:** Some indexes, like the B-tree index (commonly used), store data in sorted order. This makes operations like ORDER BY on indexed columns more efficient.
6. **Maintains Data Integrity:** Unique indexes ensure data integrity by preventing duplicate entries in the column(s) on which the index is defined.

However, it's important to note a few things about indexes:

- **Overhead:** While indexes speed up data retrieval, they add some overhead. Creating and maintaining indexes consumes additional disk space. Moreover, every time data is added or updated, the indexes also need to be updated. This can slow down write operations.
- **Selectivity:** Indexes are most effective when they are highly selective, meaning they help narrow down the results to a small subset of rows. An index on a column with many repeated values (low selectivity) might not be as effective.
- **Choosing Indexes:** It's essential to choose the right columns to index based on the queries the database will handle. Indexing every column can be counterproductive due to the overhead it introduces.

In summary, indexing is a powerful tool in the realm of databases, aiding in efficiently retrieving data. Properly designed indexes, tailored to fit the specific query patterns of a database, can vastly improve query performance.

Concurrency Control:

Concurrency control is a critical concept in database management systems (DBMS) to manage simultaneous operations without conflicting with each other, especially in a multi-user database environment.

Why is it important?

When multiple transactions are executed concurrently, the outcome might not be correct due to the overlapping steps of the transactions. For instance, two banking transactions might both try to access and modify an account balance at the same time, leading to incorrect final values. Concurrency control ensures that database consistency is maintained, and transactions are executed atomically.

Methods of Concurrency Control:

1. **Locking:** This is one of the most common methods. Resources (like rows or tables) are "locked" for a transaction's exclusive use. Other transactions can't modify a locked resource until the lock is released.
2. **Timestamping:** Every transaction is given a timestamp when it starts. The DBMS uses these timestamps to ensure older transactions get priority over newer ones, or vice versa, depending on the system.
3. **Optimistic Concurrency Control:** Transactions are executed without restrictions but are checked before committing. If conflicts are detected, the transaction might be rolled back.
4. **Multiversion Concurrency Control (MVCC):** Multiple versions of a data item are maintained. This allows read operations not to be blocked by write operations, as readers can access a previous version if the current one is being modified.

Deadlocks:

A deadlock is a situation in which two or more transactions are unable to proceed with their normal execution because each is waiting for the other to release a resource. It's a circular chain where each participant in the chain is waiting for another member to release a resource.

Example:

- Transaction A locks row 1 and needs access to row 2.
- Transaction B locks row 2 and needs access to row 1.
- Both transactions are stuck, waiting for the other to release the lock, resulting in a deadlock.

Handling Deadlocks:

1. **Prevention:** This involves ensuring that the system will never enter a deadlock state. Techniques might include ordering the way locks are acquired or using timeouts.
2. **Detection:** The system regularly checks for deadlock conditions. If found, one of the transactions is typically rolled back to break the deadlock. This might involve using wait-for graphs.
3. **Avoidance:** The system requires transactions to declare in advance what they might lock (like in the Banker's algorithm). Based on these declarations, the system decides whether to initiate or delay the transaction to avoid possible deadlocks.

Challenges:

- Detecting deadlocks can be resource-intensive.
- Rolling back transactions as a solution can be expensive and might not always be feasible, especially if the transaction has made many changes.

In a multi-user database environment, both concurrency control and deadlocks are significant concerns. Proper management ensures efficient and correct transaction processing, maintaining the database's reliability and integrity.

Question 7: Read about Database sharding and explain couple of real time examples where, why, how it this concept is used.

Database Sharding:

Database sharding is a type of database partitioning that breaks down a large database into smaller, faster, and more easily managed pieces called "shards". Each shard is a separate database instance with its own set of data, held on a separate database server. Sharding is typically done to enhance performance, scalability, and manageability of large-scale databases.

Why Use Database Sharding?

1. **Performance Enhancement:** Distributing the data reduces the load on any single server, which can result in increased performance for read and write

operations.

2. **Scalability:** As the database grows, new shards can be added to distribute the load, making it a horizontally scalable system.
3. **Geographical Distribution:** For global applications, sharding can be done based on user location to reduce latency by serving users from a nearby shard.
4. **Failover Management:** If one shard fails, it doesn't bring the whole system down. Only a fraction of users will be affected.

Real-time Examples:

1. **Social Media Platforms (e.g., Twitter):**
    - **Why:** Twitter needs to store billions of tweets from its hundreds of millions of users. A single database, no matter how powerful, would be insufficient.
    - **How:** Twitter uses sharding to distribute data. Each shard might handle data for a set of users. When a user wants to post a tweet, Twitter determines which shard that user's data lives on and directs the write operation to that shard.
    - **Result:** This approach allows Twitter to handle millions of tweets and user activities each day without significant delays or outages.
2. **E-commerce Websites (e.g., Amazon):**
    - **Why:** A global e-commerce platform like Amazon has millions of items for sale and billions of transactions. These cannot be efficiently managed on a single database.
    - **How:** Amazon might shard its database based on product categories, geographical regions, or user IDs. So, if a user from Europe accesses the platform, their requests might be directed to a shard optimized for European inventory and user data.
    - **Result:** Faster load times for product pages, efficient transaction processing, and overall improved user experience.

How is Sharding Implemented?

1. **Range-based Sharding:** Data rows are partitioned based on ranges of a certain column value. For instance, User IDs 1-1000 on Shard A, 1001-2000 on Shard B, etc.

2. **Directory-based Sharding:** A lookup service or directory determines where the data for a particular key resides. The directory maps keys to shards.
3. **Hash-based Sharding:** A hash function determines where to direct the write or read operation. This method ensures a more uniform distribution of data but can be complex to manage, especially when adding new shards.
4. **Geographical Sharding:** Data is partitioned based on geography. For example, European users might have their data on a European shard.

To implement sharding successfully, careful planning is essential. The choice of the shard key (the column on which sharding is based) is crucial, as it affects the system's scalability and performance. Backup, failover, and consistency also become more complex with sharding. However, when done right, it can immensely improve the performance and scalability of large-scale applications.