Answer to Question 2:

(a) Explain the serialization concept in Transaction. "Are all view serializable schedules conflict serializable" - Justify your answer.

Serialization Concept in Transactions:

Serialization in the context of database transactions refers to the process of ensuring that the concurrent execution of multiple transactions yields the same result as if they were executed in some serial order (one after the other). The goal is to maintain database consistency despite interleaving operations from different transactions. A schedule of operations from concurrent transactions is considered **serializable** if its effect on the database is equivalent to that of some serial execution of the same transactions.

There are two main types of serializability:

- **Conflict Serializability:** A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting[1] operations. Two operations[2] conflict if they belong to different transactions, access the same data item, and at least one of them is a write operation.[3] Non-conflicting operations can be swapped without affecting the final outcome.
- **View Serializability:** A schedule is view serializable if it is view equivalent to some serial schedule. Two schedules are view equivalent if they satisfy the[4] following conditions:
    1. For each data item, if a transaction reads the initial value in one schedule, it also reads the initial value in the other schedule.
    2. For each data item, if a transaction reads a value written by another transaction in one schedule, it also reads the value written by the same transaction in the other schedule.
    3. For each data item, if a transaction performs the final write operation in one schedule, it also performs the final write operation in the other schedule.

"Are all view serializable schedules conflict serializable?" - Justify your answer.

No, not all view serializable schedules are conflict serializable.

Justification:

Conflict serializability is a stricter condition than view serializability. Every conflict serializable schedule is also view serializable, but the converse is not necessarily true.

Consider the following non-conflict serializable schedule involving three transactions (T1, T2, T3) and a data item A, initially with value 10:

Time	T1	T2	T3
1	Read(A)		
2		Read(A)	
3	Write(A, 15)		
4		Write(A, 20)	
5			Read(A)

This schedule is **not conflict serializable** because:

- The write operation of T1 on A conflicts with the read operation of T2 on A (if T1 precedes T2) and the write operation of T2 on A (if T2 precedes T1).
- The write operation of T2 on A conflicts with the read operation of T1 on A (if T2 precedes T1) and the read operation of T3 on A (if T2 precedes T3).

However, this schedule **is view serializable** to the serial schedule T1→T2→T3:

- **Initial Read:** T1 reads the initial value of A in both schedules.
- **Intermediate Reads:** T2 reads the value written by T1 in the serial schedule (if we consider a slight modification where T2 reads after T1's write). In the concurrent schedule, T2 reads the initial value, but we can find a view equivalent serial order. T3 reads the final value of A (20) written by T2 in both the concurrent and the serial order T1→T2→T3.
- **Final Write:** T2 performs the final write to A in both the concurrent schedule and the serial order T1→T2→T3.

This example demonstrates a schedule that is view serializable but not conflict serializable. View serializability considers the final outcome and the values read by transactions, while conflict serializability focuses solely on the order of conflicting operations.

(b) What are the three data anomalies that are likely to occur as a result of data redundancy? Can data redundancy be completely eliminated in the database

approach? Why or why not?[5]

Three Data Anomalies due to Data Redundancy:

Data redundancy, the duplication of data within a database, can lead to several problems known as data anomalies. The three main types are:

1. **Insertion Anomaly:** This anomaly occurs when it becomes impossible to insert a new tuple into a relation because the primary key value is not yet associated with some of the non-key attributes. For example, if we have a table storing employee information and their department, and department details are only recorded when an employee is assigned to it, we cannot add a new department to the database if no employee is currently assigned to that department.
2. **Deletion Anomaly:** This anomaly occurs when deleting a tuple results in the unintentional loss of information about other entities. For instance, if we have a table storing employee information along with their department, and the last employee of a particular department is deleted, we might lose all information about that department if it's only stored in the employee record.
3. **Update Anomaly:** This anomaly arises when updating a data item requires updating multiple rows due to data duplication. If these updates are not performed consistently across all redundant copies, it leads to data inconsistency. For example, if an employee's address is stored in multiple records, and we update the address in only some of the records, the database will contain inconsistent information about that employee's address.

Can data redundancy be completely eliminated in the database approach? Why or why not?

No, data redundancy cannot always be completely eliminated in the database approach, and sometimes it is even intentionally introduced.

Reasons why complete elimination is not always possible or desirable:

- **Normalization Trade-offs:** While database normalization aims to minimize redundancy, achieving the highest normal forms (like 5NF) can sometimes lead to a large number of tables and complex join operations, which can negatively impact query performance. Database designers often make trade-offs between minimizing redundancy and maintaining acceptable performance levels.
- **Denormalization for Performance:** In some cases, controlled redundancy is intentionally introduced (denormalization) to improve query performance. By duplicating certain data across tables, the need for expensive join operations can

be reduced, leading to faster data retrieval for frequently accessed information. This is common in data warehousing and OLAP (Online Analytical Processing) systems.

- **Physical Storage Limitations:** While the logical design can aim for minimal redundancy, physical storage constraints or the way different database systems handle data might introduce some level of physical redundancy.
- **Derived Data:** Some redundancy is inherent in storing derived data (data calculated from other data). While it could theoretically be recalculated every time, storing it can significantly improve query performance. For example, storing the total sales amount in an order table, even though it can be calculated from the individual item prices and quantities.

In conclusion, while the relational database approach and normalization techniques strive to minimize data redundancy to avoid anomalies and ensure data integrity, complete elimination is not always practical or beneficial. Database design often involves finding a balance between minimizing redundancy and optimizing performance based on the specific requirements of the application.