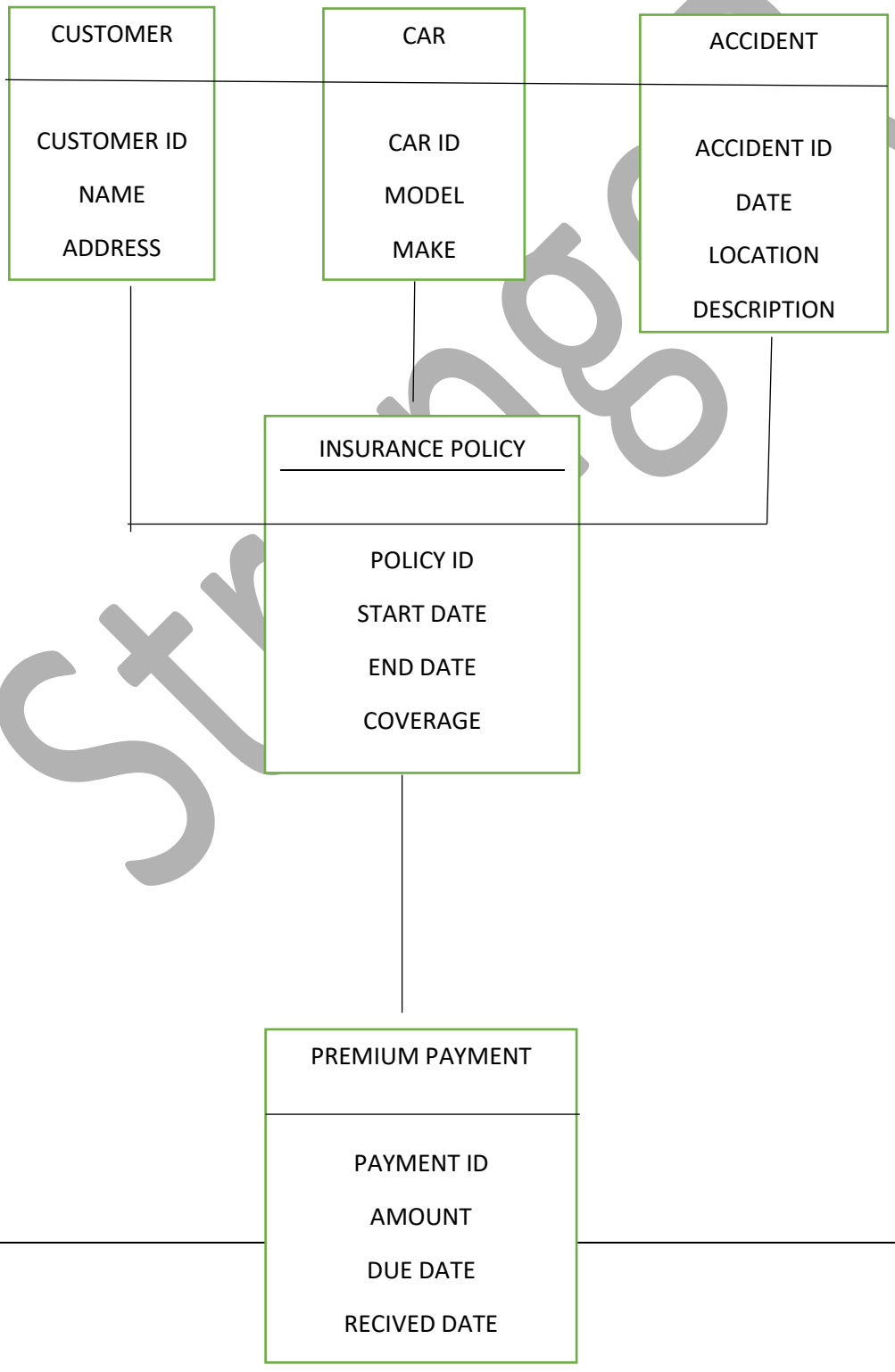
DATABASE SYSTEM DEC 2019 SOLVED BY StrangeR		
Q.NO	QUESTION	MARKS
Q.1 a)	Explain the difference between two-tier and three-tier architectures. Which is better suited for Web applications? Why? **ANSNWER:** Two-tier and three-tier architectures are both designs for distributing application logic, but they differ in their structure and functionalities. **Two-Tier Architecture:** • In a two-tier architecture, there are two main layers: the client or user interface layer and the database server layer. • The client layer is responsible for the presentation of the application and interacts directly with the database layer. • All the business logic is typically embedded in the client application, making it responsible for both the presentation and the data access logic. • It's simpler in design but can become cumbersome when dealing with a large number of users or when modifications in the logic need to be implemented. **Three-Tier Architecture:** • A three-tier architecture adds an additional layer between the client and the server layers, introducing the middle or application server layer. • The three layers are the presentation layer (client), the application server layer, and the data storage layer (database server). • The presentation layer handles the user interface, the application server layer manages the application logic and processing, and the data storage layer deals with storing and retrieving data. • This architecture separates concerns more distinctly, making it easier to scale and maintain. It also allows for more flexibility and scalability as each layer can be scaled independently. **Suitability for Web Applications:** For web applications, the three-tier architecture is generally more suitable for several reasons: 1. **Scalability:** Three-tier architectures allow for more scalable solutions. With separate layers, you can scale each layer independently based on the specific requirements. For instance, if more processing power is needed, you can scale the application server layer without affecting the other layers.	6

	2. **Maintainability:** The separation of concerns in three-tier architectures makes it easier to maintain and update the system. Changes to the presentation layer, application logic, or database layer can be done independently without affecting the entire system. 3. **Security:** By having an application server layer acting as a mediator between the user interface and the database, it's easier to implement security measures and access controls. This helps protect sensitive data from direct exposure. While both architectures have their merits, three-tier architectures tend to be more adaptable, scalable, and maintainable, making them the preferred choice for most modern web applications.	
b)	Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received. **ANSWER:** Creating an Entity-Relationship (E-R) diagram involves identifying entities, their attributes, and relationships between them. For your scenario: Entities: 1. **Customer** 2. **Car** 3. **Accident** 4. **Insurance Policy** 5. **Premium Payment** Attributes: • **Customer:** CustomerID, Name, Address, etc. • **Car:** CarID, Model, Make, etc. • **Accident:** AccidentID, Date, Location, Description, etc. • **Insurance Policy:** PolicyID, Coverage Details, Start Date, End Date, etc. • **Premium Payment:** PaymentID, Amount, Due Date, Received Date, etc. Relationships: 1. **Customer owns Car (1 to many):** A customer can own one or more cars. (1 customer to many cars)	6

2. **Car recorded in Accident (0 to many):** A car can have zero to many recorded accidents.
3. **Insurance Policy covers Car (1 to many):** An insurance policy covers one or more cars. (1 policy to many cars)
4. **Insurance Policy has Premium Payment (1 to many):** An insurance policy has one or more premium payments associated with it. (1 policy to many payments)
5. **Premium Payment has Payment Period (1 to 1):** Each payment is for a particular period.

Here's a simplified representation of the E-R diagram based on these relationships:



	This diagram depicts the relationships between entities in a car insurance system. Each entity has its attributes, and the relationships between them are described through the cardinality (how many of each entity are related to another entity)	
Q.2 b)	Consider the following database Student(name, s_no, class, major) Course(c-name, c_no, credit_hours, department) Write SQL statements to do the following update on the database schema (1) Insert a new student,<"Johnson',25,1,'Math'>,in the database. (2) Change the credit hours of course 'Data Science' to 4. (3) Delete the record for the student whose name is 'Smith' and whose student number is 17. **ANSWER:** the SQL statements to perform the requested operations: **(1) Insert a new student, <"Johnson',25,1,'Math'>, into the database: ``` INSERT INTO Student (name, s_no, class, major) VALUES ('Johnson', 25, 1, 'Math'); ``` This SQL statement will add a new record to the `Student` table with the specified values for name, student number, class, and major. **(2) Change the credit hours of the course 'Data Science' to 4: ``` UPDATE Course SET credit_hours = 4 WHERE c-name = 'Data Science'; ``` This SQL statement will update the `credit_hours` column in the `Course` table where the course name is 'Data Science', setting the credit hours to 4. (3) Delete the record for the student whose name is 'Smith' and whose student number is 17: ``` DELETE FROM Student WHERE name = 'Smith' AND s_no = 17; ``` This SQL statement will delete the record from the `Student` table where the name is 'Smith' and the student number is 17.	6

Q.3 a)	Compute the closure of the following set F of functional dependencies for relation schema r(A,B,C,D,E). A-> BC CD->E B-> D E->A List the candidate keys for R.. **ANSWER :** To compute the closure of the functional dependencies set F for relation schema r(A,B,C,D,E), we'll use Armstrong's axioms and transitive rule until no new attributes can be added. Given the functional dependencies: - A ->BC - CD ->E - B->D - E->A Let's start computing the closure of attributes for each attribute and see if we can find the candidate keys. Step 1: Compute Closure of Attributes 1. A+: A+ contains A and any other attributes it can determine transitively. - A+ = A 2. B+: B+contains B and any other attributes it can determine transitively. - B+ = BD - Using B ->D and then D ->E from CD ->E (transitive), we get B+ = BDE 3. C+: C+ contains C and any other attributes it can determine transitively. - C+ = C (No additional attributes can be determined from C 4. D+: D+ contains D and any other attributes it can determine transitively. - D+ = D (No additional attributes can be determined from D **5. E+ : E+ contains E and any other attributes it can determine transitively. - E+ = EA (Using E->A) - Then, E+ = EAB (Using A -> BC and then B -> D) Step 2: Finding Candidate Keys	6

	To find the candidate keys, we look for attributes whose closures contain all the attributes of the relation schema r(A,B,C,D,E). From the closures computed above: - A+ =A - B+ = BDE - C+ = C - D+ = D - E+ = EAB We combine attributes to check which combination forms a superkey (contains all attributes of r) and is minimal: **Possible candidate keys:** 1. AB 2. AE Both AB and AE contain all the attributes A, B, C, D, E and are irreducible (i.e., removing any attribute from them would result in a set that doesn't cover all attributes). Hence, both AB and AE are candidate keys for relation schema r(A,B,C,D,E).	
b)	Illustrating the concept of fully functional dependency, explain 2NF with example. **ANSWER:** Fully functional dependency and 2NF (Second Normal Form) are closely related concepts in database normalization. **Fully Functional Dependency: A fully functional dependency occurs when a field in a table is functionally dependent on the entire primary key, not just a part of it. For example, consider a table `Employees` with columns `EmployeeID`, `ProjectID`, and `ProjectName`, where `EmployeeID` is the primary key. - If `ProjectName` is functionally dependent on `EmployeeID` and `ProjectID` together, i.e., for each combination of `EmployeeID` and `ProjectID`, there is only one corresponding `ProjectName`, then it's a fully functional dependency. - However, if `ProjectName` is dependent only on `ProjectID` and not on both `EmployeeID` and `ProjectID` together, then it's not a fully functional dependency. **2NF (Second Normal Form): 2NF deals with removing partial dependencies in a relation by ensuring that all non-prime attributes are fully functionally dependent on the entire primary key. A table is in 2NF if: 1. It is in 1NF (First Normal Form). 2. All non-prime attributes are fully functionally dependent on the entire primary key.	6

Example:

Let's consider a table `EmployeeProjects`:

EmployeeID	ProjectID	ProjectName	EmployeeName
1	101	ProjectA	John
1	102	ProjectB	John
2	101	ProjectA	ALICE

In this table:
- `(EmployeeID, ProjectID)` is the composite primary key.
- `ProjectName` is functionally dependent on `ProjectID` (and not on the entire primary key) because for each `ProjectID`, `ProjectName` remains the same regardless of the `EmployeeID`.
- `EmployeeName` is functionally dependent on `EmployeeID` but not on the entire primary key.

To make it 2NF compliant:
- We separate the data into two tables:
  - `Projects` (ProjectID, ProjectName)
  - `Employees` (EmployeeID, EmployeeName)
- The `EmployeeProjects` table becomes:

EmployeeID	ProjectID
1	101
1	102
2	101

Now, `ProjectName` is only in the `Projects` table, and `EmployeeName` is only in the `Employees` table. This arrangement ensures that all non-prime attributes are fully functionally dependent on the primary key in their respective tables, satisfying 2NF.

Q.4 a) Let relations r1 (A, B,C) and r2 (C,D,E) have the following properties: r1 has 20,000 tuples, r2 has 45,000 tuples, 25 tuples of r1 fit on one block, and 30 tuples of r2 fit on one block. Estimate the number of blocks transfers and seeks required, using each of the following join strategies for r1∞ r2: (r1 Natural Join r2)

1. Nested-loop join.

2. Block nested-loop join.

ANSWER:

To estimate the number of block transfers and seeks required for performing the join operation r1 ∞ r2 (where r1 is the natural join with r2) using different join strategies, we'll consider the properties of the relations and the join algorithms.

6

Given:
- r1has 20,000 tuples, and 25 tuples fit in one block.
- r2 has 45,000 tuples, and 30 tuples fit in one block.

1. Nested-Loop Join:

In a nested-loop join, for each tuple in \(r1\), all tuples in \(r2\) are scanned to find matching tuples, resulting in \(r1\) being the outer relation.

For r1 ∞ r2 using nested-loop join:

- **Number of block transfers:**
  - For $r1$: $\lceil \frac{20000}{25} \rceil = 800$ blocks
  - For $r2$: $\lceil \frac{45000}{30} \rceil = 1500$ blocks
- **Total number of block transfers:**
  - $800 \times 1500 = 1,200,000$ block transfers

- **Number of seeks:**
  - Each block of r1 will need to be read multiple times for each block of r2.
  - Seeks required will be significant due to repeated scans of r1 for each block of r2.

2. Block Nested-Loop Join:

Block nested-loop join involves reading blocks of one relation and performing nested-loop joins with blocks of the other relation.

For $r1 \bowtie r2$ using block nested-loop join:

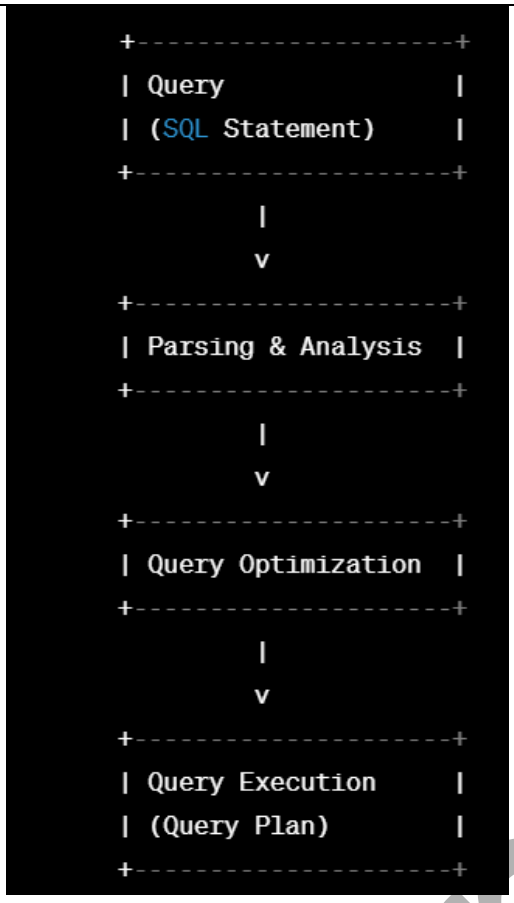
- **Number of block transfers:**
  - For $r1$: $\lceil \frac{20000}{25} \rceil = 800$ blocks (unchanged)
  - For $r2$: $\lceil \frac{45000}{30} \rceil = 1500$ blocks (unchanged)
- **Total number of block transfers:**
  - $800 \times 1500 = 1,200,000$ block transfers (unchanged)

- **Number of seeks:
  - The block nested-loop join doesn't reduce the number of seeks compared to the nested-loop join, as it still requires multiple scans of blocks for each block of the other relation.

Both the nested-loop join and block nested-loop join strategies result in the same number of block transfers and seek operations for the given relations r1 and r2 due to the nature of these join algorithms, where one relation is scanned multiple times for each block of the other relation.

b)	Explain Query processing? Explain various steps in query processing with the help of neat sketch.	

	ANSWER: Query processing is the process of executing a query in a database system. It involves a series of steps that transform a query written in a high-level language (like SQL) into a sequence of low-level operations that the database system can execute to retrieve the required data. The main steps in query processing include: 1. **Parsing and Analysis:** • **Parsing:** The query is checked for syntax errors and parsed into a parse tree or syntax tree. • **Semantic Analysis:** The query is checked for semantic correctness, including checking for the existence of tables and columns, permissions, etc. 2. **Optimization:** • **Query Optimization:** The query optimizer generates alternative execution plans for the query and chooses the most efficient one based on cost estimation, considering indexes, join algorithms, access paths, etc. 3. **Query Execution:** • **Query Plan Generation:** The chosen execution plan is converted into a series of operations (e.g., scans, joins, sorts) represented in a query execution plan. • **Query Execution:** The database executes the query by following the steps outlined in the query execution plan. Here's a neat sketch illustrating the steps in query processing:	6



- **Parsing & Analysis:** The query is parsed and checked for syntax and semantic correctness. If errors are found, an error message is returned.
- **Query Optimization:** The optimizer generates different execution plans and selects the most efficient one based on cost estimation and statistics.
- **Query Execution:** The chosen execution plan is executed by the database engine, fetching data from storage, performing necessary operations like filtering, joining, sorting, and finally returning the result to the user.

Each step in query processing is crucial for efficient and accurate retrieval of data from a database system. Optimizing these steps leads to faster query processing and improved database performance.

Q. 5 a) Construct a B+-tree for the following set of key values:

(2, 3, 5, 7, 11, 17 , 19, 23, 29, 31)

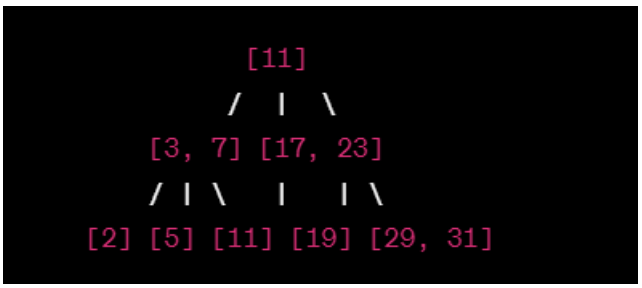
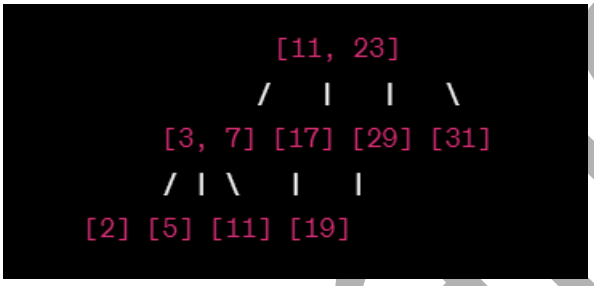
Assume that the tree is initially empty and values are added in ascending order. Construct

B+ tree for the cases where the number of pointers that will fit in one node is as follows:

i. Four

ii. Six

6

	ANSWER: constructing B+ trees for the given set of key values, assuming the maximum number of pointers that will fit in one node is four and six. **B+ Tree with Four Pointers Per Node:** **Step 1:** Creating the B+ tree with four pointers per node: 1. Start with an empty root. 2. Add keys sequentially: 2, 3, 5, 7, 11, 17, 19, 23, 29, 31. The B+ tree would look like this:  **B+ Tree with Six Pointers Per Node:** **Step 1:** Creating the B+ tree with six pointers per node: 1. Start with an empty root. 2. Add keys sequentially: 2, 3, 5, 7, 11, 17, 19, 23, 29, 31. The B+ tree would look like this:  **Explanation:** • In both cases, the B+ tree maintains the keys in sorted order. • With four pointers per node, the tree has more levels compared to the tree with six pointers per node. • A larger number of pointers per node results in a shallower tree and potentially faster search times but requires more keys to fill a node before a split occurs. • The structure and organization of the B+ tree change based on the maximum number of pointers allowed in each node, affecting the tree's height and fanout at each level.	
b)	Define ordered indices. Differentiate between Dense and sparse indices with suitable example. **ANSWER:** Ordered indices are data structures used in databases to improve the efficiency of searching, accessing, and retrieving data. They are based on a sorted ordering of key values and enable faster lookup of records based on those keys. **Dense Indices:** **Definition:** Dense indices contain an entry for every search key value in the database. • **Example:** Consider a simple scenario with a student database where the primary key is the student ID. In a dense index, there would be an entry for every student	6

	ID in the index. If a student with ID 101, 102, 103, and so on exists, the index will have entries for each of these IDs.
	• **Characteristics:**
	  • Require more storage space as they store an entry for every key value.
	  • Generally quicker for searching since they directly point to the location of the record in the data file.
	  • Efficient for equality searches but may not be suitable for range queries.
	Sparse Indices:
	Definition: Sparse indices do not contain entries for every possible search key value.
	• **Example:** Continuing with the student database, if the IDs are sparse and not all values exist (e.g., student IDs 101, 103, 107 exist, but 102, 104, 105 are missing), a sparse index will only have entries for the existing IDs. So, it will contain entries for 101, 103, and 107 but not for 102, 104, and 105.
	• **Characteristics:**
	  • Occupy less space as they only store entries for existing key values.
	  • Slower for direct searches since they may not directly point to the location of the record; they might need to traverse other entries to locate the desired record.
	  • Can be advantageous for range queries because they store fewer entries, and scanning fewer index entries might be quicker for range operations.
	Difference between Dense and Sparse Indices:
	• **Density of Entries:**
	  • Dense indices have an entry for every key value, while sparse indices only have entries for existing key values.
	• **Storage Space:**
	  • Dense indices require more storage space as they contain entries for every key value, whereas sparse indices occupy less space as they only store entries for existing key values.
	• **Search Efficiency:**
	  • Dense indices are generally faster for direct searches since they directly point to the record's location.
	  • Sparse indices might be slower for direct searches as they may require traversing more entries to locate the desired record.
	The choice between using a dense or sparse index depends on the nature of the data, the query patterns, and the trade-offs between storage space and search efficiency.	

Q.6	Write short note on following: (1) ACID properties of transaction (2) View serializable schedule **ANSWER:-** **(1) ACID Properties of Transaction:** ACID is an acronym representing the four essential properties of a transaction in a database system:	12

- **Atomicity:** Transactions are atomic, meaning they are indivisible and execute fully or not at all. If any part of a transaction fails, the entire transaction is rolled back to its initial state, ensuring that the database remains consistent.
- **Consistency:** Transactions transition the database from one consistent state to another consistent state. Each transaction must abide by all the rules, constraints, and validations defined in the database schema.
- **Isolation:** Transactions execute independently of each other, ensuring that the intermediate states of one transaction are not visible to other concurrent transactions. This prevents interference or data corruption caused by multiple transactions running concurrently.
- **Durability:** Once a transaction is committed, its changes are permanently saved in the database even in the event of system failure. The changes become durable and should not be lost, ensuring that the data remains consistent despite crashes or errors.

(2) View Serializable Schedule:

A schedule in database transactions refers to the order in which various transactions are executed. A schedule is considered view serializable if it produces the same results as some serial execution of the transactions.

- **View Serializability:** It ensures that the final result (state of the database) of concurrent execution of multiple transactions should be equivalent to the result obtained when the transactions are executed serially, one after another.
- **Conflict Serializable vs. View Serializable:** View serializability is a stronger condition than conflict serializability. While conflict serializability focuses on the ordering of conflicting operations between transactions, view serializability considers the overall effect of all operations in each transaction.
- **Testing for View Serializability:** To determine if a schedule is view serializable, you can use techniques like the view serializability matrix, where transactions are represented as nodes, and dependencies between transactions are represented as edges in a graph.

Ensuring view serializability is crucial to maintain data integrity and consistency in database systems, especially in scenarios where multiple transactions are executed concurrently. It ensures that the results obtained from concurrent execution are equivalent to some serial execution of those transactions, maintaining consistency and correctness of the database.