LAB ASSIGNMENT

NAME- RAVI SHEKHAR
ROLL - 22CS3075

Q1. Analyzing Serializability

To construct the precedence graph, we need to identify conflicting operations between transactions. Conflicting operations are those where one transaction performs a write operation on a data item that another transaction reads or writes.

For the given schedule:

T1: R(A), W(A)

T2: R(B), W(B)

T3: R(A), W(A), R(B), W(B)

We can identify the conflicting operations:

- T1 writes A, conflicting with T3's read and write of A.
- T2 writes B, conflicting with T3's read and write of B.

Hence, the precedence graph looks like this:

```
T1 -----> T3

 |         |

 V         V

T2 -----> T3
```

This graph has no cycles, indicating that the schedule is conflict serializable. Since there are no cycles, any serial order will preserve the read and write dependencies, making it view serializable as well.

Q.2. Simulating Transactions in SQL

First, we'll create a SQLite database with two tables representing data items A and B. Then, we'll write SQL scripts to execute the given schedule as transactions. We'll run these transactions under different isolation levels to observe if the final state of the database varies.

Here's a step-by-step approach:

1. Create a SQLite database with tables for data items A and B:

```
CREATE TABLE A (

   id INTEGER PRIMARY KEY,

   value INTEGER

);

CREATE TABLE B (

   id INTEGER PRIMARY KEY,

   value INTEGER

);
```

2. Write SQL scripts to execute the transactions from the given schedule:

```
-- Transaction 1

BEGIN TRANSACTION;
```

```
UPDATE A SET value = 1 WHERE id = 1;
COMMIT;

– Transaction 2
BEGIN TRANSACTION;
UPDATE B SET value = 2 WHERE id = 1;
COMMIT;

– Transaction 3
BEGIN TRANSACTION;
UPDATE A SET value = 3 WHERE id = 1;
UPDATE B SET value = 4 WHERE id = 1;
COMMIT;
```

3. Run the transactions under different isolation levels. You can use the `BEGIN TRANSACTION` statement with appropriate isolation levels like `SERIALIZABLE`, `READ COMMITTED`, etc., before executing each transaction. For example:

```
– Set isolation level to SERIALIZABLE
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
– Execute transaction 1
– Execute transaction 2
– Execute transaction 3
COMMIT;

– Set isolation level to READ COMMITTED
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
-- Execute transaction 1
-- Execute transaction 2
-- Execute transaction 3
COMMIT;
```

4. Observe the final state of the database after running the transactions under different isolation levels. If the final state varies, it indicates non-serializable executions.

Q.3. Implementing a Scheduler in Python

To implement the scheduler in Python, we'll follow these steps:

1. Parse the input schedule to extract transaction operations.
2. Construct a precedence graph based on conflicting operations.
3. Check if the graph has cycles using graph-based algorithms.
4. If the graph has no cycles, output "Conflict Serializable: Yes".
5. Perform a topological sort to generate an equivalent serial schedule.

Here's a Python implementation:

```
from collections import defaultdict

class Scheduler:

    def __init__(self):

        self.graph = defaultdict(list)
```

```
def add_operation(self, operation):

    transaction, action = operation.split("(")

    action = action.strip(")").split(",")

    if action[0] == 'R':

        self.graph[transaction].append(action[1])

    elif action[0] == 'W':

        for key in self.graph.keys():

            if action[1] in self.graph[key]:

                self.graph[key].append(transaction)



def has_cycle(self, node, visited, stack):

    visited[node] = True

    stack[node] = True



    for neighbor in self.graph[node]:

        if not visited[neighbor]:
                if self.has_cycle(neighbor, visited, stack):
                    return True
            elif stack[neighbor]:
                return True
        stack[node] = False
        return False


    def is_conflict_serializable(self):
        visited = {key: False for key in self.graph}
        stack = {key: False for key in self.graph}


        for node in self.graph:
            if not visited[node]:
                if self.has_cycle(node, visited, stack):
                    return False
        return True
```

```
def topological_sort_util(self, node, visited, stack):
    visited[node] = True

    for neighbor in self.graph[node]:
        if not visited[neighbor]:
            self.topological_sort_util(neighbor, visited, stack)

    stack.append(node)

def topological_sort(self):
    visited = {key: False for key in self.graph}
    stack = []

    for node in self.graph:
        if not visited[node]:
                self.topological_sort_util(node, visited, stack)


        return stack[::-1]


    def get_equivalent_serial_schedule(self):
        if not self.is_conflict_serializable():
            return None


        serial_schedule = self.topological_sort()
        return serial_schedule




def main():
    input_schedule = [
        "R1(A)", "W1(A)",
        "R2(B)", "W2(B)",
        "R3(A)", "W3(A)",

        "R3(B)", "W3(B)"

    ]



    scheduler = Scheduler()



    for operation in input_schedule:

        scheduler.add_operation(operation)



    conflict_serializable = scheduler.is_conflict_serializable()

    equivalent_serial_schedule = scheduler.get_equivalent_serial_schedule()



    print("Conflict Serializable:", "Yes" if conflict_serializable else "No")

    if conflict_serializable:

        print("Equivalent Serial Schedule:", ", ".join(equivalent_serial_schedule))

if __name__ == "__main__":
    main()
```

This implementation takes the input schedule, constructs a precedence graph, checks for cycles, and determines if the schedule is conflict serializable. If it is, it generates an equivalent serial schedule using topological sorting.