Transaction

Q1: Specify which of the following schedules could be generated by

a) Two phase locking scheduler
b) Rigorous two-phase locking scheduler

If the schedule could be generated by 2PL, specify when transactions would acquire locks and release locks. If not, explain why the lock request will fail and hence the schedule would not be generated.

1. w_1 (x) w_2(x) w_1(y) w_2(y) c_1 c_2
2. w_1(x) w_2(x) w_2(y) w_1(y) c_1 c_2
3. w_1(x) r_2(x) w_3(y) w_2(y) c_1 c_2 c_3

In the schedules above, the subscript refers to the id of the transactions, that is, an operation w_1(x) is a write operation by transaction T1 on data item x.

w_1(x), w_2(x), w_1(y), w_2(y), c_1, c_2

1a)

Two-Phase Locking (2PL) protocol feasible:

1. **T1** acquires an exclusive lock on **x** and then on **y**.
2. **T1** writes to **x** and releases the exclusive lock on **x**.
3. **T2** acquires the exclusive lock on **x** and writes to **x**.
4. **T1** writes to **y** and then releases the exclusive lock on **y**.
5. **T2** acquires the exclusive lock on **y** and writes to **y**.
6. Both **T1** and **T2** commit, after which **T2** releases the locks on **x** and **y**.

1b)
This schedule fails under Rigorous 2PL because **T2** cannot acquire locks on **x** or **y** while **T1** is holding them. Rigorous 2PL requires all locks to be held until the transaction commits. However, in this schedule, **T2** needs to acquire locks before **T1** commits, which violates the protocol.

w_1(x), w_2(x), w_2(y), w_1(y), c_1, c_2

1a)

Two-Phase Locking (2PL) protocol not feasible:

1. **T1** acquires the lock on **x** and performs a write.

2. **T2** then acquires the lock on **x** and performs a write.
3. After **T2** writes to **x**, it needs to acquire a lock on **y**. However, this creates a conflict since **T1** holds the lock on **y**.
4. If **T1** releases the lock on **y** for **T2**, **T1** would not be able to write to **y** afterward, violating the 2PL protocol.

1b)
This schedule is also not possible under Rigorous 2PL. Since rigorous 2PL requires transactions to hold all their locks until commit, **T2** would not be able to acquire the lock on **x** while **T1** holds it, and **T1** would not release it until it completes. The same conflict arises as with standard 2PL.

w_1(x), r_2(x), w_3(y), w_2(y), c_1, c_2, c_3

1a)

Two-Phase Locking (2PL) protocol feasible:

1. **T1** acquires an exclusive lock on **x**, writes to **x**, and then releases the lock.
2. **T2** acquires a shared lock on **x** and reads from **x**.
3. **T3** acquires an exclusive lock on **y**, writes to **y**, and releases the lock.
4. **T2** then acquires the lock on **y**, writes to **y**, and releases all locks.
5. Finally, **T1**, **T2**, and **T3** commit.

1b)

This schedule cannot be executed under Rigorous 2PL. Rigorous 2PL mandates that transactions hold all locks until they commit, preventing locks from being released for other transactions to proceed. Therefore, **T1**, **T2**, and **T3** would block each other, making the schedule infeasible.

Indexing

Q2: Consider a relation Employee(
ssn INT PRIMARY KEY,
name char(30),
salary REAL CHECK (salary BETWEEN 50K AND 250K)
department CHAR(50))

Let us consider the following indices:

I1: CREATE INDEX salary_index ON Employee (salary)

I2: CREATE INDEX composite_index ON Employee(name, department)

I3: CREATE INDEX ssn_index ON Employee (ssn)

Now consider the following queries:

Q1: SELECT * from Employee where ssn = 123 AND name LIKE "Mike%"

Q2: SELECT * from Employee where name = "Jim"

Q3: SELECT * from Employee where department = "CS"

Q4: SELECT * from Employee where salary > 51000 AND salary < 249000

For each of the queries above, explain which index, if any, would be useful in reducing the execution time of the query.

Query Q1: SELECT * from Employee where ssn = 123 AND name LIKE "Mike%"

Use **INDEX 2 and 3**, as these indexes include the name and ssn attributes, which are both required in this query.

Query Q2: SELECT * from Employee where name = "Jim"

Use **INDEX 2**, as it includes the name attribute in the Employee table

Query Q3: SELECT * from Employee where department = "CS"

No index is effective in reducing execution time

Query Q4: SELECT * from Employee where salary > 51000 AND salary < 249000

Use **INDEX 1**, as it contains the salary attribute in the Employee table.

Object-Relational Database System

Q3: Consider the following sequence of SQL statements

```
create type Person
   (ID varchar(20) primary key,
    name varchar(20),
    address varchar(20))
    ref from(ID);
create table people of Person;


create type Department (
    dept_name varchar(20),
    head ref(Person) scope people);
create table departments of Department

insert into people values ('12345', 'Alice', '123 Wonderland Ave');
insert into people values ('23456', 'Bob', '456 Nowhere Blvd');
insert into people values ('34567', 'Charlie', '789 Imaginary St');

insert into departments values ('CS', '12345');
insert into departments values ('Math', '23456');
insert into departments values ('Physics', '34567');

create type Project (
   proj_name varchar(20),
   lead ref(Person) scope people,
   department ref(Department) scope departments
);
create table projects of Project;

insert into projects values ('AI Research', '23456', 'CS');
insert into projects values ('Quantum Computing', '34567', 'Physics');
```

Q3. What will be the output if we now execute the following SQL query

```
select proj.proj_name as ProjectName,
      proj.lead->name as LeadName,
      proj.department->dept_name as DepartmentName,
    proj.department->head->address as HeadAddress
from projects proj;
```

Output:

ProjectName	LeadName	DepartmentName	HeadAddress
AI Research	Bob	CS	123 Wonderland Ave
Quantum Computing	Charlie	Physics	789 Imaginary St

Q4: Transform the last query above into a relational query considering the reference of tables will become ID when creating the table and you need to do the join between tables rather than simply using "->".
Example of "reference of tables will become ID when creating the table":

```
create type Department (
   dept_name varchar(20),
   head ref(Person) scope people);
```

Will become

```
create type Department (
   dept_name varchar(20),
   people_id varchar(20) );
```

```
SELECT
       proj.proj_name AS ProjectName,
       people_lead.name AS LeadName,
       departments.dept_name AS DepartmentName,
       people_head.address AS HeadAddress
FROM
       projects proj
JOIN
       people people_lead ON proj.lead = people_lead.ID
JOIN
       departments ON proj.department = departments.dept_name
JOIN
       people people_head ON departments.people_id = people_head.ID;
```