Assignment 3

Started: 29 Mar at 14:57

Quiz instructions

This assignment examines concurrent transaction executions under three different isolation levels.

This is an **individual assignment** to be submitted by each student.

The assignment consists of two parts (A and B), where each part considers a different database schema and transaction workload. Each part consists of five 1-mark questions, and the total score for this assignment is 10 marks. The assignment is due on **April 14 (Friday, 2359)**, and it is to be submitted using **Canvas Quizzes (Assignment 3).**

Late submission penalty: There will be a late submission penalty of 1 mark per day up to a maximum of 3 days. If your assignment is late by more than 3 days, it will not be graded and you will receive 0 marks for the assignment.

Isolation Levels

The assignment considers three different isolation levels (RC, RR & SI).

The first isolation level is **Read Committed (RC)** under the lock-based concurrency control protocol:

1. A short-duration shared lock is requested right before each read operation, and the lock is released right after the read operation.
2. A long-duration exclusive lock is requested right before each insert/update operation.
3. Long duration locks are released only after the transaction has committed/aborted.
4. A transaction may acquire further locks after releasing shared locks before it commits/aborts.
5. Lock upgrades and downgrades are not allowed.

The second isolation level is **Repeatable Read (RR)** under the lock-based concurrency control protocol:

1. A long-duration shared lock is requested right before each read operation.
2. A long-duration exclusive lock is requested right before each insert/update operation if the transaction is not holding any shared lock on the object. Otherwise, the transaction will request for a lock upgrade.
3. All locks are released only after the transaction has committed/aborted.
4. Lock downgrades are not allowed.

The third isolation level is the **Snapshot Isolation (SI)** protocol. Unlike Serializable Snapshot Isolation (SSI), SI schedules may not be MVSS.

For the two lock-based isolation levels (RC & RR), deadlocks are managed using **deadlock detection**. Each deadlock is resolved by randomly aborting one of the transactions in the detected deadlock cycle.

Part A: Database Schema & Transaction Workload

This part consists of five questions (question 1 to 5), and the questions are based on the following database schema for a flight booking application.

```
create table Planes (
        pid integer primary key,
        num_seats integer
);

create table Flights (
        fid integer primary key,
        pid integer references Planes,
        num_reserved_seats integer
);


create table Reservations (
        pa_id integer,
        fid integer references Flights,
        seat_number integer,
        primary key (pa_id, fid)
);
```

- The **Planes** relation stores information about the seat capacity (i.e., **num_seats**) of planes (identified by **pid**). The seats on a plane are numbered from 1 to n if its value of num_seats is n.
- The **Flights** relation stores information about each flight: the unique identifier for the flight (i.e, **fid**), the plane used for the flight (i.e, **pid**), and the number of seats that have been reserved for the flight (i.e., **num_reserved_seats**).
- The **Reservations** relation stores information about the flight(s) reserved by passengers: the unique identifier for a passenger (i.e, **pa_id**), the unique identifier for the flight being reserved (i.e, **fid**), and the assigned seat number for the passenger on the flight (i.e., **seat_number**).
- The only index created on each table is the index on the table's primary key; and the only specified integrity constraints are the primary/foreign key constraints specified in the table definitions.
- To reserve a seat on a flight for a passenger, a **Reserve transaction** is executed which accepts two input parameters (**flight_num** and **passenger_id**). The first statement retrieves the flight record corresponding to flight_num and stores its pid and num_reserved_seats values into the local variables p and nr, respectively. The second statement retrieves the plane record with a pid value of p and stores its num_seats value into the local variable ns. The third statement compares nr and ns; if nr < ns, the database is updated as follows: (1) the number of reserved seats for the flight is updated to nr+1, and (2) a new reservation record is created for the passenger and flight with an assigned seat number of nr+1.
- Note that if any of a transaction's update/insert statement executions violates a table's primary/foreign key constraint, the transaction will be aborted.

```
-- Reserve transaction for Part A
-- Input parameters: (1) flight_num and (2) passenger_id
-- Local variables: p, nr, ns
begin;

select pid, num_reserved_seats into p,  nr
from Flights
where fid = flight_num;

select num_seats into ns
from Planes
where pid = p;

if (nr < ns) then
        update Flights
        set num_reserved_seats = nr + 1
        where fid = flight_num;

        insert into Reservations values (passenger_id, flight_num, nr+1);
        commit;
else
        rollback;
end if;
```

- The **transaction workload** consists of multiple instances of Reserve transaction executing concurrently.

Part B: Database Schema & Transaction Workload

This part consists of five questions (question 6 to 10), and the questions are based on a modified version of the database schema used in Part A.

```
create table Planes (
        pid integer primary key,
        num_seats integer
);

create table Flights (
        fid integer primary key,
        pid integer references Planes
);


create table Reservations (
        pa_id integer,
        fid integer references Flights,
        seat_number integer,
        primary key (pa_id, fid)
);
```

- The schema for tables **Planes** and **Reservations** are the same as those in Part A.
- The only difference is the schema for **Flights** where the **num_reserved_seats** is no longer explicitly stored.
- As in Part A, the **transaction workload** consists of multiple instances of Reserve transaction executing concurrently. However, the Reserve transaction is a modified version of that used in Part A.
- As before, the modified **Reserve transaction** accepts two input parameters (**flight_num** and **passenger_id**). The first statement retrieves the flight record corresponding to flight_num and

stores its pid value into the local variable p. The second statement retrieves the plane record with a pid value of p and stores its num_seats value into the local variable ns. The third statement retrieves all the reservation records for the flight to determine the last assigned seat number for the flight (i.e., maximum seat_number among the retrieved records) and store the maximum value into the local variable nr; if there are no matching flight records, the value of nr is set to 0. The fourth statement compares nr and ns; if nr < ns, a new reservation record is created for the passenger and flight with an assigned seat number of nr+1.

- As before, if any of a transaction's update/insert statement executions violates a table's primary/foreign key constraint, the transaction will be aborted.

```
-- Reserve transaction for Part B
-- Input parameters: (1) flight_num and (2) passenger_id
-- Local variables: p, nr, ns
begin;

select pid into p
from Flights
where fid = flight_num;

select num_seats into ns
from Planes
where pid = p;

select coalesce(max(seat_number),0) into nr
from Reservations
where fid = flight_num;

if (nr < ns) then
        insert into Reservations values (passenger_id, flight_num, nr+1);
        commit;
else
        rollback;
end if;
```

Question 1 — 1 pts

For Part A's transaction workload, it is possible for **RC** to produce a schedule (where all transactions commit) that is not conflict serializable.

- (●) True
- ( ) False

Question 2 — 1 pts

For Part A's transaction workload, it is possible for a deadlock to occur under **RC**.

- ○ True
- ◉ False

Question 3 — 1 pts

For Part A's transaction workload, it is possible for **RR** to produce a schedule (where all transactions commit) that is not conflict serializable.

- ○ True
- ◉ False

Question 4 — 1 pts

For Part A's transaction workload, it is possible for a deadlock to occur under **RR**.

- ◉ True
- ○ False

Question 5 — 1 pts

For Part A's transaction workload, It is possible for **SI** to produce a schedule (where all transactions commit) that is not multiversion view serializable.

- ○ True
- ◉ False

Question 6 — 1 pts

For Part B's transaction workload, it is possible for **RC** to produce a schedule (where all transactions commit) that is not conflict serializable.

- (●) True
- ( ) False

Question 7 — 1 pts

For Part B's transaction workload, it is possible for a deadlock to occur under **RC**.

- ( ) True
- (●) False

Question 8 — 1 pts

For Part B's transaction workload, it is possible for **RR** to produce a schedule (where all transactions commit) that is not conflict serializable.

- ( ) True
- (●) False

Question 9 — 1 pts

For Part B's transaction workload, it is possible for a deadlock to occur under **RR**.

- (●) True
- ( ) False

Question 10 — 1 pts

For Part B's transaction workload, it is possible for **SI** to produce a schedule (where all transactions commit) that is not multiversion view serializable.

- ( ) True
- (●) False

Saved at 2:22 [Submit quiz]