CS186 Week 10 - Discussion Notes Solution

Compare locking and optimistic concurrency control (OCC). List two advantages of locking and two advantages of OCC.

Advantages of OCC:
(1) avoids the possibility of deadlock
(2) avoids the overhead of lock management
(3) can allow a larger space of possible schedules than locking (depending on the locking scheme)
(4) an interesting advantage of OCC occurs when clients are not collocated; in this case, OCC only requires synchronization between clients at commit time, vs. synchronization for every access in the case of locking.

Advantages of locking:
(1) can offer better throughput under heavy contention (much fewer transaction abort + restarts => less wasted work)
(2) avoids the need to track read and write sets

Thinking about race conditions: what can be the query results of the two transactions below executed concurrently if we didn't make isolation guarantees?

Transaction 1

```
INSERT INTO ratings VALUES('larry', 9.5);
INSERT INTO ratings VALUES('kevin',2.5);
INSERT INTO ratings VALUES('george', 2.3);
```

Transaction 2

```
SELECT MIN(rating) FROM ratings;
SELECT MAX(rating) FROM ratings;
SELECT COUNT(*) FROM ratings WHERE rating=9.5;
```

Ratings

Name	**Rating**
'John'	9.4
'Foo'	4.3
'Bar'	2.8
'Eric'	9.4
'Jack'	7.7

Answer:

2.8, 9.4, 0
2.8, 9.4, 1
2.8, 9.5, 1

2.5, 9.5, 1
2.3, 9.5, 1

What's one case where snapshot isolation would not guarantee serializability?
Answer: Transactions need to obtain write locks for only the tuples it has modified. Let's say Joe has two bank accounts B1 and B2 and both bank accounts have $50 in them. Let's say there is the constraint that B1 + B2 >= 0. Joe then concurrently withdraws $100 from both bank accounts. T1 withdraws 100 from B1 and T2 withdraws 100 from B2. B1 now equals -50 and B2 now equals -50. We see now that B1 + B2 = -100, but T1 sees B1=-50 and B2=50 and can therefore commit (since it can see a snapshot of the database). T2 sees B1=50 and B2=-50 and similarly can commit.

What values can the program below print? What can we do to make the program below deterministic?

```
public class TestConcurrent {
   public static void main(String[] args) {
      Incrementer incr = new Incrementer();
      Thread t1 = new Thread(incr);
      Thread t2 = new Thread(incr);
      t1.start();
      t2.start();
      try {
         t1.join();
         t2.join();
      } catch (InterruptedException e) {
         System.err.println("Oops something wrong");
      }
      System.out.println(incr.count);
   }

   private static class Incrementer implements Runnable {
      int count = 0;
      public void run() {
         for (int i = 0; i < 3; i++) {
            increment();
         }
      }
      public void increment() {
         count = count + 1;
      }
```

```
    }
}
```

Answer

This program can print any integer between 3 and 6, inclusive. This program, would ideally print out 6 since you're incrementing count 6 times (3 times each thread).
count = count + 1 breaks down to (read count) (modify the count that was read) (write count). This looks okay at first if count = count + 1 was an atomic operation. One case where this might go wrong:
Thread 1 and thread 2 read count=2 at the same time.
Thread 1 then incremented count = count + 1 and wrote count = 3.
Thread 2 increments count it read to count = count + 1 and write count = 3.
Both threads incremented once, but in this example we only incremented count by 1.

Can make deterministic by adding synchronized block. synchronized(this) block around critical section or make the increment method synchronized.