COMP9315 22T1 DBMS Implementation

Exercises 09

Transaction Processing: Concurrency, Recovery

1. Consider the following transaction *T1*: `R(X), R(X)`

   a. Give an example of another transaction *T2* that, if run concurrently to transaction *T1* without some form of concurrency control, could interfere with *T1* to produce unrepeatable reads. Show the sequence of operations which would cause the problem.

   Answer:

   Even a transaction as simple as *T2*: `W(X)` is sufficient to cause unrepeatable reads. If *T2* runs concurrently with *T1* and if the `W(X)` operation occurs between the two `R(X)` operations, then *T1* sees a different value of `X` on each of the read operations. The following schedule shows the problem:

   ```
   T1:  R(X)        R(X)
   T2:       W(X)
   ```

   b. Explain how the application of strict two-phase locking would prevent the problem described in your previous answer.

   Answer:

   In the case of two-phase locking the transactions would be re-written as:

   ```
   T1: ReadLock(X)  R(X)  R(X)  Unlock(X)

   T2: WriteLock(X)  W(X)  Unlock(X)
   ```

   If *T1* starts first and acquires the ReadLock on `X`, then *T2* will be unable to proceed until *T1* has completed its two reads (it cannot acquire the WriteLock while another transaction holds a ReadLock). If *T2* starts first and acquires the WriteLock on `X`, then *T1* will be unable to proceed until *T2* has completed (it cannot acquire a ReadLock while another transaction holds a WriteLock).

   The following two schedules are the only possible ones that can occur (where "`.....`" indicates a delay while waiting for a lock):

   ```
   Schedule 1:

   T1:  ReadLock(X)  R(X)  R(X)  Unlock(X)
   T2:               ......................  WriteLock(X)  W(X)  Unlock

   Schedule 2:

   T1:               ..................  ReadLock(X)  R(X)  R(X)  Unlock(X
   T2: WriteLock(X)  W(X)  Unlock(X)
   ```

2. SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

   a. Describe which of the following phenomena can occur at each of the four SQL isolation levels: dirty read, unrepeatable read, phantom problem.

   Answer:

   For Read Uncommitted, all three may occur.
   For Read Committed, only unrepeatable read and the phantom problem may occur.
   For Repeatable Read, only the phantom problem may occur.
   For Serializable, none of the problems can occur.

   b. Why does the access mode of a transaction matter?

   Answer:

   If all transactions are READ-ONLY mode, then the isolation level doesn't matter, since none of the problems noted above can occur. As soon as even one transaction is READ/WRITE access mode, then the above problems *may* occur, and an appropriate isolation-level needs to be used to avoid any of the above problems that are undesirable.

3. Draw the precedence graph for the following schedule:

```
T1:      R(A) W(Z)                C
T2:                R(B) W(Y)        C
T3: W(A)                      W(B)     C
```

Answer:

It has an edge from T3 to T1 (because of A) and an edge from T2 to T3 because of B.

This gives: `T2 --> T3 --> T1`

4. [Based on RG Ex.17.2] Consider the following incomplete schedule *S*:

```
T1: R(X) R(Y) W(X)           W(X)
T2:               R(Y)            R(Y)
T3:                    W(Y)
```

   a. Determine (by using a precedence graph) whether the schedule is serializable

Answer:

The precedence graph has an edge, from *T1* to *T3*, because of the conflict between `T1:R(Y)` and `T3:W(Y)`. It also has an edge, from *T2* to *T3*, because of the conflict between the first `T2:R(Y)` and `T3:W(Y)`. It also has an edge, from *T3* to *T2*, because of the conflict between `T3:W(Y)` and the second `T2:R(Y)`. The edges between *T2* and *T3* form a cycle, so the schedule is not conflict-serializable.

b. Modify *S* to create a complete schedule that is conflict-serializable

Answer:

One possibility would be to move the `W(Y)` operation in *T3* to the end (i.e. make it the last operation). An alternative would be to move the second `R(Y)` in *T2* to just before the `W(Y)` operation in *T3*.

5. Is the following schedule conflict serializable? Show your working.

```
T1:                R(X) W(X) W(Z)           R(Y) W(Y)
T2: R(Y) W(Y) R(Y)                W(Y) R(X)          W(X) R(V) W(V)
```

Answer:

As above, the working for this question involves constructing a precedence graph, based on conflicting operations, and looking for cycles.

In this case there's a conflict between `T1:R(X)` and `T2:W(X)`, giving a graph edge from `T1` to `T2`. There's also a conflict between `T2:R(Y)` and `T1:W(Y)`, giving a graph edge from `T2` to `T1`. This means the graph has a cycle, so the schedule is not serializable.

6. [Based on RG Ex.17.3] For each of the following schedules, state whether it is conflict-serializable and/or view-serializable. If you cannot decide whether a schedule belongs to either class, explain briefly. The actions are listed in the order they are scheduled, and prefixed with the transaction name.

a. `T1:R(X) T2:R(X) T1:W(X) T2:W(X)`
b. `T1:W(X) T2:R(Y) T1:R(Y) T2:R(X)`
c. `T1:R(X) T2:R(Y) T3:W(X) T2:R(X) T1:R(Y)`
d. `T1:R(X) T1:R(Y) T1:W(X) T2:R(Y) T3:W(Y) T1:W(X) T2:R(Y)`
e. `T1:R(X) T2:W(X) T1:W(X) T3:W(X)`

Answer:

The techniques used to determine these solutions: for conflict-serializablility, draw precedence graph and look for cycles; for view-serializablility, apply the definition from lecture notes.

a. not conflict-serializable, not view serializable

b. conflict-serializable, view serializable (view equivalent to T1,T2)
c. conflict-serializable, view serializable (view equivalent to T1,T3,T2)
d. not conflict-serializable, not view serializable
e. not conflict-serializable, view serializable (view equivalent to T1,T2,T3)

7. Recoverability and serializability are both important properties of concurrent transaction schedules. They are also orthogonal. Serializability requires that the schedule be equivalent to some serial ordering of the transactions. Recoverability requires that each transaction commits only after all of the transactions from which is has read data have also committed.

   Using the following two transactions:

```
T1:  W(A)  W(B)  C        T2:  W(A)  R(B)  C
```

   give examples of schedules that are:

   a. recoverable and serializable
   b. recoverable and not serializable
   c. not recoverable and serializable

   Answer:

   a. the following schedule is both recoverable and serializable

```
T1:  W(A)  W(B)             C
T2:             W(A)  R(B)     C
```

      The update operations of the schedules are serial, and therefore serializable. For recoverability, the only read is `R(B)` in T2, and T2 does not commit until after T1 has committed.

   b. the following schedule is recoverable but not serializable

```
T1:        W(A)  W(B)       C
T2:  W(A)             R(B)     C
```

      There is a cycle in the dependency graph T2--A->T1 and T1--B->T2, so it's not serializable. However, T1 commits before T2, so it's recoverable for the same reasons as (a).

   c. the following schedule is not recoverable but is serializable

```
T1:  W(A)  W(B)                C
T2:             W(A)  R(B)  C
```

      It is clearly serializable for the same reason as (a). However, since T2 commits before T1 we could end up in the following scenario: T2 commits completely but the system crashes before T1 does. After recovery, T2 would remain committed, but T1 would be rolled back because it has no log entry. Since the result of T2 depends on T1's completion, we would have a contradiction; T2 should not be able to complete if T1 does not complete.

8. ACR schedules avoid the potential cascading rollbacks that can make recoverable schedules less than desirable. Using the transactions from the previous question, give an example of an ACR schedule.

   Answer:

   The following schedule is clearly serializable. However, since no reads are performed on data modified by uncommitted transactions, there is also no chance of cascading rollback.

```
T1:  W(A)  W(B)        C
T2:              W(A)     R(B)  C
```

9. Consider the following two transactions:

```
     T1              T2
------------     ------------
read(A)          read(B)
A := 10*A+4      B := 2*B+3
write(A)         write(B)
read(B)          read(A)
B := 3*B         A := 100-A
write(B)         write(A)
```

   a. Write versions of the above two transactions that use two-phase locking.

   Answer:

   The basic idea behind two-phase locking is that you take out all the locks you need, do the processing, and then release the locks. Thus two-phase implementations of `T1` and `T2` would be:

```
     T1              T2
------------     ------------
write_lock(A)    write_lock(B)
read(A)          read(B)
A := 10*A+4      B := 2*B+3
write(A)         write(B)
write_lock(B)    write_lock(A)
read(B)          read(A)
B := 3*B         A := 100-A
write(B)         write(A)
unlock(A)        unlock(B)
unlock(B)        unlock(A)
```

   b. Is there a non-serial schedule for *T1* and *T2* that is serializable? Why?

   Answer:

   No. It's not possible. The last operation in *T1* is `write(B)`, and the last operation in *T2* is `write(A)`. *T1* starts with `read(A)` and *T2* starts with `read(B)`. Therefore, in any

serializable schedule, we would require that either `read(A)` in *T1* should be after `write(B)` in *T2* or `read(B)` in *T2* should be after `write(B)` in *T1*.

c. Can a schedule for *T1* and *T2* result in deadlock? If so, give an example schedule. If not, explain why not.

Answer:

Yes. Consider the following schedule (where `L(X)` denotes taking an exclusive lock on object `X`):

```
T1: L(A) R(A)      W(A) L(B) ...
T2:           L(B)           R(B) W(B) L(A) ...
```

10. What is the difference between quiescent and non-quiescent checkpointing? Why is quiescent checkpointing not used in practice?

Answer:

In quiescent checkpointing, the system must wait until there are no active transactions, then flush the transaction log and write a checkpoint record. All transactions before the checkpoint are complete, and do not need to be considered further in any subsequent recovery.

In non-quiescent checkpointing, the system marks a checkpoint period via a pair of start-checkpoint and end-checkpoint records. The checkpoint covers a period in time during which active transactions may have made changes to the database. Recovery to checkpoints is more complicated, and must take account of which transactions were active before and after the checkpoint started, and which ones completed during the checkpoint period.

Because real database systems are heavily used and must have high availability. The first point (heavy use) means that there is never have any point in time when there are no transactions executing. The second point (high availability) means that we can't afford to block all new transactions while the system runs checkpointing.

11. Consider the following sequence of undo/redo log records:

```
<START T> ; <T,A,10,11> ; <T,B,20,21> ; <COMMIT T>
```

Give all of the sequences of "events" that are legal according to the rules of undo/redo logging. An "event" consists of one of: writing to disk a block containing a given data item, and writing to disk an individual log record.

Answer:

In this example, there are five events, which we shall denote:

- A: database element A is written to disk.

- B: database element B is written to disk.
- LA: the log record for A is written to disk.
- LB: the log record for B is written to disk.
- C: the commit record is written to disk.

The only constraints that undo/redo logging requires are that LA appears before A, and LB appears before B. Of course the log records must be written to disk in the order in which they appear in the log: LA,LB,C. The eight orders consistent with these constraints are:

- LA,A,LB,B,C
- LA,A,LB,C,B
- LA,LB,A,B,C
- LA,LB,A,C,B
- LA,LB,B,A,C
- LA,LB,C,A,B
- LA,LB,B,C,A
- LA,LB,C,B,A

12. Consider the following sequence of undo/redo log records from two transactions T and U:

```
<START T> ; <T,A,10,11> ; <START U> ; <U,B,20,21> ;
<T,C,30,31> ; <U,D,40,41> ; <COMMIT U> ; <T,E,50,51>;
<COMMIT T>
```

Describe the actions of the recovery manager, if there is a crash and the last log record to appear on disk is:

a. `<START U>`

b. `<COMMIT U>`

c. `<T,E,50,51>`

d. `<COMMIT T>`

You may assume that there is an `<END CKPT>` record in the log immediately before the start of transaction T, so that we do not need to worry about the log any further back than the start of T.

Answer:

Each recovery session these commences with a scan forward from the most recent checkpoint (just before start of T), to determine which transactions had started and which had committed since the checkpoint.

a. Since neither T nor U is committed, we need to undo all of their actions. We move backwards through the log to the most recent checkpoint. Since U had only just started, we find no actions by it to undo. However, we do find an update record for T, so we reset the value of A to 10.

b. Since U is committed, we redo its actions, setting B to 21 and D to 41. Then, since T is uncommitted, we undo its actions from the end moving backwards; we reset C to 30 and A to 10.

c. This is similar to the previous part, except that we also reset E to 50.

d. In this case, both transactions have finished, and so we need to redo all of their actions, setting A to 11, B to 21, C to 31, D to 41 and E to 51. There are no incomplete transactions, so there is no need for an undo pass.