COMP207 Lab Exercises

Tutorial 3 - Week 5

The exercises below provide the opportunity to practice the concepts and methods discussed during the lectures. Don't worry if you cannot solve all the exercises during the lab session, but try to tackle at least one or two of them. If at some point you do not know how to proceed, you could review the relevant material from the lecture notes and return to the exercise later.

Logging and Recovery

Exercise 1 (Exercise 17.2.4/17.2.5 in [2])**.** The following is a sequence of undo-log records written by two transactions, T_1 and T_2:

<START T_1>
<T_1, X, 10>
<START T_2>
<T_2, Y, 20>
<T_1, Z, 30>
<T_2, U, 40>
<COMMIT T_2>
<T_1, V, 50>
<COMMIT T_1>

(a) Describe the action of the recovery manager, including changes to both the database and the log on disk, if there is a system failure and the last log record to appear on disk is:

(i) <START T_2>

(ii) <COMMIT T_2>

(iii) <T_1, V, 50>

(iv) <COMMIT T_1>

(b) For each of the situations (i)–(iv) in (a), describe what values written by T_1 and T_2 *must* appear on disk? Which values *might* appear on disk?

Solutions:

(a) **(i)** In this case, the log on disk looks like this:

<START T_1>
<T_1,X,10>
<START T_2>

The recovery manager processes the log in reverse order, starting from the last log record (<START T_2>) to the first one (<START T_1>). No action is required by the recovery manager when it sees <START T_2>, so it moves on to <T_1,X,10>. Since the recovery manager hasn't seen a <COMMIT T_1> or an <ABORT T_1> record before it arrived at <T_1,X,10>, it changes the value of X on disk back to 10, and moves on to the next log record in line, which is <START T_1>. No action is required by the recovery manager when it sees <START T_1>.

As a final step, the recovery manager appends to the log on disk the following two log records: <ABORT T_1> and <ABORT T_2>. The final log on disk is:

<START T_1>
<T_1,X,10>
<START T_2>
<ABORT T_1>
<ABORT T_2>

This finishes the work of the recovery manager.

(ii) In this case, the log on disk looks like this:

<START T_1>
<T_1,X,10>
<START T_2>
<T_2,Y,20>
<T_1,Z,30>
<T_2,U,40>
<COMMIT T_2>

As before, the recovery manager processes the log in reverse order, starting from the last log record (<COMMIT T_2>) to the first one (<START T_1>):

Step	Current log record	Recovery manager's action
1	<COMMIT T_2>	Remember for later steps that T_2 has finished.
2	<T_2,U,40>	Do nothing, because T_2 has finished.
3	<T_1,Z,30>	Overwrite the value of Z on disk by 30. This is because T_1 hasn't finished (we haven't seen <COMMIT T_1> or <ABORT T_1>).
4	<T_2,Y,20>	Do nothing.
5	<START T_2>	Do nothing.
6	<T_1,X,10>	Overwrite the value of X on disk by 10.
7	<START T_1>	Do nothing.

As a final step, the recovery manager appends to the log on disk the log record <ABORT T_1>. The final log on disk is:

<START T_1>
<T_1,X,10>
<START T_2>
<T_2,Y,20>
<T_1,Z,30>
<T_2,U,40>
<COMMIT T_2>
<ABORT T_1>

This finishes the work of the recovery manager.

(iii) In this case, the log on disk looks like this:

<START T_1>
<T_1,X,10>
<START T_2>
<T_2,Y,20>
<T_1,Z,30>
<T_2,U,40>
<COMMIT T_2>
<T_1,V,50>

The action of the recovery manager on this log is quite similar to the one in (ii). The only difference is the first step, where it processes <T_1,V,50>:

Step	Current log record	Recovery manager's action
1	<T_1,V,50>	Overwrite the value of V on disk by 50. This is because T_1 hasn't finished (we haven't seen <COMMIT T_1> or <ABORT T_1>).
2	<COMMIT T_2>	Remember for later steps that T_2 has finished.
3	<T_2,U,40>	Do nothing, because T_2 has finished.
4	<T_1,Z,30>	Overwrite the value of Z on disk by 30.
5	<T_2,Y,20>	Do nothing.
6	<START T_2>	Do nothing.
7	<T_1,X,10>	Overwrite the value of X on disk by 10.
8	<START T_1>	Do nothing.

As a final step, the recovery manager appends to the log on disk the log record <ABORT T_1>. The final log on disk is:

<START T_1>
<T_1,X,10>
<START T_2>
<T_2,Y,20>
<T_1,Z,30>
<T_2,U,40>
<COMMIT T_2>
<T_1,V,50>
<ABORT T_1>

This finishes the work of the recovery manager.

(iv) In this case, all log records were written to disk, so the recovery manager will not perform any changes on disk, neither to the database items nor to the log:

Step	Current log record	Recovery manager's action
1	<COMMIT T_1>	Remember for later steps that T_1 has finished.
2	<T_1,V,50>	Do nothing, because T_1 has finished.
3	<COMMIT T_2>	Remember for later steps that T_2 has finished.
4	<T_2,U,40>	Do nothing, because T_2 has finished.
5	<T_1,Z,30>	Do nothing.
6	<T_2,Y,20>	Do nothing.
7	<START T_2>	Do nothing.
8	<T_1,X,10>	Do nothing.
9	<START T_1>	Do nothing.

This finishes the work of the recovery manager.

(b) **(i)** The value of X written by T_1 might appear on disk, because we see a corresponding log entry. But there is no guarantee that the value was written to disk, because the log does not contain a <COMMIT T_1> record. No other values could have been written to disk.

(ii) The situation for X and Z is as in (i): the values of X and Z written by T_1 might have been written to disk, but there is no guarantee for that. Since the log contains a <COMMIT T_2> record, we know that the values of Y and U written by T_2 must have been written to disk.

(iii) Same as in (ii), except that the value for V written by T_1 might appear on disk, but that's not guaranteed.

(iv) All values written by T_1 and T_2 must appear on disk.

Exercise 2 (Exercise 19.1.2/19.1.3 in [2])**.** Consider the following schedules:

- S_1: $r_1(X)$; $r_2(Y)$; $w_1(Y)$; $w_2(Z)$; $r_3(Y)$; $r_3(Z)$; $w_3(U)$
- S_2: $r_1(X)$; $w_1(Y)$; $r_2(Y)$; $w_2(Z)$; $r_3(Z)$; $w_3(U)$

- S_3: $r_2(X)$; $r_3(X)$; $r_1(X)$; $r_1(Y)$; $r_2(Y)$; $r_3(Y)$; $w_2(Z)$; $r_3(Z)$
- S_4: $r_2(X)$; $r_3(X)$; $r_1(X)$; $w_1(Y)$; $r_3(Y)$; $w_2(Z)$; $r_3(Z)$

(a) Suppose that each of the schedules is followed by an abort operation for transaction T_1. Tell which transactions need to be rolled back.

(b) Now suppose that all three transactions commit and write their commit record on the log immediately after their last operation. However, a crash occurs, and a tail of the log was not written to disk before the crash and is therefore lost. Tell, depending on where the lost tail of the log begins:

(i) What transactions could be considered uncommitted?

(ii) Are any dirty reads created during the recovery process? If so, what transactions need to be rolled back?

(iii) What additional dirty reads could have been created if the portion of the log lost was not a tail, but rather some portions in the middle?

Solutions:

(a)
- S_1: T_1 and T_3 have to be rolled back: T_1 because it might have written some values to disk, and T_3 because it reads a value (Y) that was written before by T_1. No other transaction needs to be rolled back.
- S_2: All three transactions have to be rolled back, because T_2 reads a value (Y) written by T_1, and T_3 reads a value (Z) written by T_2.
- S_3: No transaction needs to be rolled back. Note that T_1 does not need to be rolled back, because it did not write any values.
- S_4: Same as for S_1.

(b) No complete solution provided. This is essentially a combination of Exercise 1 and Exercise 2 (a). Let us say the first transaction in the schedule is the one who finishes first, the second transaction in the schedule is the one who finishes second, and the third transaction in the schedule is the one who finishes last. For each schedule, we distinguish the following possibilities for the time of the crash:

1. before the commit record of the first transaction in the schedule reaches disk;
2. after the commit record of the first transaction in the schedule reaches disk, but before the commit record of the second transaction in the schedule reaches disk;
3. after the commit record of the second transaction in the schedule reaches disk, but before the commit record of the third transaction in the schedule reaches disk;
4. after the commit record of the third transaction in the schedule reaches disk.

For part (i) of the question: In case 1, all transactions would be considered uncommitted. In case 2, only the second and the third transaction in the schedule would be considered uncommitted. In case 3, only the third transaction in the schedule would be considered uncommitted. And in case 4, no transaction would be considered uncommitted.

For part (ii): If a transaction reads an item that was written by a transaction that is considered uncommitted, then that is a dirty read. In this case, that transaction needs to be rolled back. All transactions who depend on that transaction also need to be rolled back and so on (cascading rollback).

For part (iii): No solution provided. For discussion.

Recoverable, Cascadeless, and Strict Schedules

Exercise 3 (Exercise 20.24 in [1])**.** For each of the following schedules, determine if the schedule is (A) recoverable, (B) cascadeless, (C) strict, (D) non-recoverable. Try to determine the strictest recoverability condition that each schedule satisfies.

(a) $S_1: r_1(X);\ r_2(Z);\ r_1(Z);\ r_3(X);\ r_3(Y);\ w_1(X);\ c_1;\ w_3(Y);\ c_3;\ r_2(Y);\ w_2(Z);\ w_2(Y);\ c_2$

(b) $S_2: r_1(X);\ r_2(Z);\ r_1(Z);\ r_3(X);\ r_3(Y);\ w_1(X);\ w_3(Y);\ r_2(Y);\ w_2(Z);\ w_2(Y);\ c_1;\ c_2;\ c_3$

(c) $S_3: r_1(X);\ r_2(Z);\ r_3(X);\ r_1(Z);\ r_2(Y);\ r_3(Y);\ w_1(X);\ c_1;\ w_2(Z);\ w_3(Y);\ w_2(Y);\ c_3;\ c_2$

Solutions:

Schedule	Recoverable?	Cascadeless?	Strict?	Strictest Condition
S_1	yes	yes	yes	strict
S_2	no	no	no	non-recoverable
S_3	yes	yes	no	cascadeless

a) S1 is both recoverable and cascadeless because every transaction commits right after it writes some items. It is also strict because there is no write to or read from an item before the last transaction that wrote that item has committed.

b) S2 is neither recoverable, nor cascadeless, nor strict because T2 has read item Y that was written before by T3 and T2 has committed before T3.

c) S3 is both recoverable and cascadeless because no transaction reads items that were written by an uncommitted transaction. However, it is not strict because T2 writes Y before T3 commits.

Exercise 4 (Exercise 19.1.1 in [2])**.** What are all the ways to insert lock operations (of the simple lock type only), unlock operations, and commit operations into

$$r_1(X);\ r_1(Y);\ w_1(X);\ w_1(Y)$$

so that the transaction T_1 is:

(a) Two-phase locked, and strict two-phase locked.

(b) Two phase locked, but not strict two-phase locked.

Solutions:

(a) The locking operation for X must occur before $r_1(X)$, the commit operation must come immediately after $w_1(Y)$, followed by two unlock operations for X and Y (which can come in any order). The locking operation for Y can be inserted anywhere before $r_1(Y)$. This leads to the six combinations:

- $l_1(Y)$; $l_1(X)$; $r_1(X)$; $r_1(Y)$; $w_1(X)$; $w_1(Y)$; c_1; $u_1(X)$; $u_1(Y)$
- $l_1(X)$; $l_1(Y)$; $r_1(X)$; $r_1(Y)$; $w_1(X)$; $w_1(Y)$; c_1; $u_1(X)$; $u_1(Y)$
- $l_1(X)$; $r_1(X)$; $l_1(Y)$; $r_1(Y)$; $w_1(X)$; $w_1(Y)$; c_1; $u_1(X)$; $u_1(Y)$
- $l_1(Y)$; $l_1(X)$; $r_1(X)$; $r_1(Y)$; $w_1(X)$; $w_1(Y)$; c_1; $u_1(Y)$; $u_1(X)$
- $l_1(X)$; $l_1(Y)$; $r_1(X)$; $r_1(Y)$; $w_1(X)$; $w_1(Y)$; c_1; $u_1(Y)$; $u_1(X)$
- $l_1(X)$; $r_1(X)$; $l_1(Y)$; $r_1(Y)$; $w_1(X)$; $w_1(Y)$; c_1; $u_1(Y)$; $u_1(X)$

(b) As in (a), the locking operation for X must occur before $r_1(X)$, and the locking operation for Y can be inserted anywhere before $r_1(Y)$. The unlocking operation for X must come after $w_1(X)$, and the unlocking operation for Y must come after $w_1(Y)$. To be not strict 2PL, one of the unlocking operations must come before the commit operation.

References

[1] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson Education, 7th edition, 2016.

[2] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Pearson Education, 2nd edition, 2009.