Tutorial 4

Assignment 1

Remember that the deadline for assignment 1 is Friday at 5pm (i.e. the 29th)!!!

This week's topic

We looked at locks and logs

More precisely, we saw 2PL, simple and more advanced locks and 3 logging schemes: Undo, Redo, Undo/redo and when and how they could be used

2PL: all locks are before every unlock in each transaction

Shared and exclusive locks: Shared is only for read and exclusive can do everything

Shared, exclusive and upgrade locks: Shared and upgrade are only for read, exclusive for everything, but you can't get an exclusive lock on an item if you have shared lock for it

Intention locks: Locks on different levels (like table, block and row level)

Undo is used to directly ensure atomicity (and if some requirements are satisfied also durability)

Redo is used to directly ensure durability (and if some requirements are satisfied also atomicity)

Undo/redo is used to ensure both

Question 1

Consider the following two transactions:

T_1: $read(X)$; $read(Y)$; $Y := X + 2Y$; $write(Y)$;

and

T_2: $read(Y)$; $read(X)$; $X := Y + 2X$; $write(X)$;

There are multiple ways to add in shared lock, exclusive lock and unlock operations to the transactions so that they obey 2PL. Can you find a way to add in locks such that some schedules on them lead to a deadlock and another way so that no schedule on those leads to a deadlock?

Solution for Question 1

A possible solution that could lead to a deadlock would be:

T_1 : s-lock(X);read(X);x-lock(Y);read(Y);Y := X +2Y;write(Y);unlock(X);unlock(Y);

and

T_2 : s-lock(Y);read(Y);x-lock(X);read(X);X := Y +2X;write(X);unlock(Y);unlock(X);

Those can lead to deadlocks, e.g. see the following part of a schedule:

$S: sl_1(Y) r_1(Y); sl_2(X); r_2(X); ?$

A possible solution that can't lead to a deadlock would be:

T_1 : x-lock(X);x-lock(Y);read(X);read(Y);Y := X +2Y;write(Y);unlock(Y);unlock(X);

and

T_2 : x-lock(X);x-lock(Y);read(Y);read(X);X := Y +2X;write(X);unlock(Y);unlock(X);

There are no deadlocks, since the first transaction to move can keep on moving (while the other transaction must wait until the first finishes).

Question 2

(Exercise 17.2.4/17.2.5 in Database Systems: The complete book). 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?

Solution for Question 2.a.i

In this case, the log on disk looks like this:

<START T_1>

<T_1,X,10>

<START T_2>

Undo logging works in reverse order…

<START T_2>: No action is required by the recovery manager

<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

<START T_1>: No action is required by the recovery manager

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.

Solution for Question 2.a.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>

Undo logging works in reverse order…

<COMMIT T_2>: remember that T_2 has finished

<T_2,U,40>: T_2 has finished so skip

<T_1,Z,30>: T_1 has **not** finished, so overwrite Z with 30

<T_2,Y,20>: T_2 has finished so skip

<START T_1>: No action required

<T_1,X,10>: T_1 has **not** finished, so overwrite X with 10

<START T_2>: No action required

As a final step, the recovery manager appends to the log on disk the following log record: <ABORT T_1>. We end with:

<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>

Solution for Question 2.a.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>

Undo logging works in reverse order…

<T_1,V,50>:T_1 has **not** finished, so overwrite V with 50

<COMMIT T_2>: remember that T_2 has finished

<T_2,U,40>: T_2 has finished so skip

<T_1,Z,30>: T_1 has **not** finished, so overwrite Z with 30

<T_2,Y,20>: T_2 has finished so skip

<START T_1>: No action required

<T_1,X,10>: T_1 has **not** finished, so overwrite X with 10

<START T_2>: No action required

As a final step, the recovery manager appends to the log on disk the following log record: <ABORT T_1>. We end with:

<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>

Solution for Question 2.a.vi

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>

<COMMIT T_1>

Undo logging works in reverse order…

<COMMIT T_1>: remember that T_1 has finished

<T_1,V,50>: T_1 has finished, so skip

<COMMIT T_2>: remember that T_2 has finished

<T_2,U,40>: T_2 has finished, so skip

<T_1,Z,30>: T_1 has finished, so skip

<T_2,Y,20>: T_2 has finished so skip

<START T_1>: No action required

<T_1,X,10>: T_1 has finished, so skip

<START T_2>: No action required

Solution for Question 2.b

b.i: X might have been written to disk, but without <COMMIT T_1> we can't be sure

b.ii: X and Z might have been written to disk, but without <COMMIT T_1> we can't be sure. U and Y has been written to disk, since we see <COMMIT T_2>

b.iii: V, X and Z might have been written to disk, but without <COMMIT T_1> we can't be sure. U and Y has been written to disk, since we see <COMMIT T_2>

b.vi: U, V, X, Y and Z have been written to disk, since we see <COMMIT T_1> and <COMMIT T_2>

Question 3

(Exercise 19.1.2/19.1.3 in Database Systems: the complete book). 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?

Solution to question 3

3.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

3.b: No complete solution provided. Let us say transactions finishes so that transaction 1 finishes before 2, before 3. 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.

Solution to question 3 continued

For part i. of the question: In case 1. “before the commit record of the first transaction in the schedule reaches disk;”, all transactions would be considered uncommitted.

In case 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;”only the second and the third transaction in the schedule would be considered uncommitted

In case 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;”, only the third transaction in the schedule would be considered uncommitted

In case 4, “after the commit record of the third transaction in the schedule reaches disk”, 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.