ACS Theory Assignment 2

Anders Holst, wlc376

2020-12-14

1 Concurrency Control Concepts

- *1. Show a schedule that is view-serializable, but not conflict-serializable.*

I try to devise as simple a schedule as possible that still meets the specifications:

```
View-serializable, non-conflict-serializable schedule

T1:      W(A)
T2: R(A)      W(A)
T3:                W(A)
```

The schedule shown is *not* conflict-serializable because there is a cycle in its precedence graph (`T2 -> T1 -> T2`).

Consider then the dependency graph for the schedule. Since `T2` and `T1` make the first read and update of `A`, respectively, `T1` must execute before `T2`, producing the dependency `T1 -> T2`.

Since `T3` makes the last update of `A`, it must come after all other transactions which update `A` (ie. both `T1` and `T2`), producing the dependencies `T1 -> T3` and `T2 -> T3`. Since there are no write-before-read conflict to be considered, the dependency graph is:

```
T1 ----------> T2
\              /
 \---> T3 <---/
```

There are no cycles in the dependency graph, so the schedule must be view-serializable.

- *2. Show a schedule that is conflict-serializable, but which could not have been generated by an S2PL scheduler.*

Consider below schedule in which two transactions access the same variable `B`, and where one also reads a different variable. `S(V)` and `X(V)` denote shared and exclusive locks on some variable `V`. Assume that locks are implicitly unlocked in `C`.

```
Conflict-serializable, non-S2PL schedule

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

The precedence graph for this schedule contains simply one edge from `T1` to `T2`, corresponding to the write-after-write conflict on `B`.

However, it could not have been produced by an S2PL scheduler, since `T1` cannot release its exclusive lock on `B` before it has committed.

- *3. Show a schedule that could be generated by an S2PL scheduler, but not by a C2PL scheduler.*

Consider below schedule in which two transactions access two variables `A` and `B` (once again, locks are implicitly released in `C`):

```
S2PL, non-C2PL schedule. Locks are implicitly unlocked in C.

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

This could have been generated by an S2PL scheduler with the inserted locks. It is a valid S2PL schedule since `T1` does not have to acquire the shared lock on `B` before it can make the read to `A`. However, it is *not* a valid C2PL schedule for the exactly the opposite reason: the read of `A` before `T1` has the shared lock on `B` is not legal in C2PL.

- *4. Show a schedule that could be generated by a C2PL scheduler, but not by an S2PL scheduler.*

In the below schedule, `U(V)` denotes the explicit unlocking of some lock `V`, but `C` still implicitly unlocks all locks still held at commit time.

```
C2PL, non-S2PL schedule

T1: S(A) S(B) R(A) U(A)             R(B) C
T2:                     X(A) W(A) C
```

Conservative 2PL requires that `T1` gathers shraed locks on both `A` and `B` before begings its read of `A`; however, `T1` is allowed to release its lock on `A` before reading `B`, and thus `T2` is able to grab the lock on `A` before `T1` has read `B`.

However, it is precisely for this reason that this schedule could *not* have been generated by an S2PL scheduler; here, `T1` would not have been able to release its lock on `A` before it had also locked and read `B`.

---

2 More concurrency control

Consider the below two schedules:

```
Schedule 1
Ta: R(X)                        W(X) C
Tb:      R(X) W(Y)      C
Tc:               R(X)   W(Y) C
```

```
Schedule 2
Ta: R(X)                             W(Z) C
Tb:      R(Z) R(Y) W(Y) C
Tc:                      R(X) W(X) C
```

- *1. Could schedule 1 have been generated by an S2PL scheduler?*

Yes! With lock conversion, a transaction can upgrade its shared lock to an exclusive lock of the same resource if no other transaction holds a shared lock of that resoruce. If we assume lock conversion is possible, then schedule 1 could have been generated by a strict 2PL scheduler as shown below:

```
Schedule 1
Ta: S(X) R(X)                                          X(X) W(X) C
Tb:           S(X) X(Y) R(X) W(Y)           C
Tc:                              S(X) R(X)   X(Y) W(Y) C
```

Notice that `Ta` first acquires a shared lock of `X`, and only upgrades this to an exclusive lock when necessary. In the meantime `Tb` is able to both grab a shared lock of `X` and release it again.

- *2. Could schedule 1 have been generated by an 2PL scheduler?*

Yes! Any schedule that can be produced by a strict 2PL scheduler can also be produced by a 2PL scheduler, so the same reasoning as before can be used here.

- *3. Could schedule 2 have been generated by a C2PL scheduler?*

No! `Ta` cannot make its read of `X` before it has acquired all the locks it will need, but it will need an exclusive lock for `Z`, and this will prevent `Tb` to acquire a shared lock of `Z`.

- *4. Could schedule 2 have been generated by a Kung-Robinson scheduler?*

Short answer: no!

Slightly longer answer: In Kung-Robinson, transactions are assigned ID's at the end of their read phases, and when validating transactions, any given transaction is compared to transactions which ended their read phases later.

In the snippet below, I have inserted vertical bars to annotate where I assume read phases to end:

```
1  Schedule 2
2  Ta: R(X) |                                W(Z) C
3  Tb:        R(Z) R(Y) | W(Y) C
4  Tc:                          R(X) | W(X) C
```

This would indicate that I should start by validating `Ta`.

Validating `Ta`

Both `Tb` and `Tc` begin before `Ta` ends, so test 1 fails.

Both `Tb` and `Tc` also start their write phases before `Ta` completes, so test 2 fails aswell.

`Ta` completes its read phase before both `Tb` and `Tc` do, but the condition that `WriteSet(Ta)` must not overlap with `ReadSet(Tx)` or `WriteSet(Tx)` holds for neither `Tx = Tb` or `Tx = Tc`.

Since we have already invalidated `Ta`, there is no reason to examine the other transactions.

3 Recovery Concepts

- *1. In a force/steal system, is it necessary to implement a scheme for redo?*

If pages are forced to disk before a transaction is allowed to commit, rather than flushing when it is convenient, then there is never a need to redo transactions that have already committed.

- *What about undo?*

With *steal*, transactions might push pages to disk which have not been committed yet; if this is the case, then this must be undone.

- *2. What is the difference between volatile and non-volatile storage, and what types of failures are survived by each?*

Put shortly, volatile storage is only active when powered on, whereas non-volatile storage retains its state when the power is on. Examples are main memory and disk memory, respectively.

Volatile storage is lost on crashes, whereas non-volatile storage is not; however, both are vulnerable to hardware failure (bit flipping, regular wear and tear, etc). Especially harddisks with moving parts are vulnerable to failures from wear and tear.

- *3. In WAL, which are the two situations in which the log tail is forced to disk?*

First, when pages are updated, a log record for that update is forced to disk before the page is actually modified in memory.

Second, before a given transaction is allowed to commit, all log records for that transaction must be forced to disk.

- *Why are log forces necessary, and why are they sufficient for durability?*

One big reason they are necessary is the no-force policy, since the log may be the only record of certain committed transactions having even taken place.

They are sufficient for durability because they record all *intended* mutations before they are carried out, and that is all that is needed for redo/undo during recovery.

---

4 ARIES

- *1. The transaction and dirty page tables after the analysis phase.*

```
/----------------------------\
|      Transaction Table     |
|----------------------------|
| xact id | status | last_LSN |
|---------+--------+----------|
| T1      | U      | 9        |
\----------------------------/

/--------------------\
|  Dirty Page Table  |
|--------------------|
| page id | rec_LSN  |
|---------|----------|
| P2      | 3        |
| P1      | 4        |
| P5      | 5        |
\--------------------/
```

The last log record for `T2` was a commit, so it would have been temporarily inserted into the transaction table during the analysis phase, but afterwards it would have been removed and an `end` record written to the log.

- *2. The sets of winner and loser transactions.*

The set of winner transaction is simply the singleton set `{T3}`, while the loser transactions is `{T1, T2}`, since these were still active at the time of crash.

- *3. The LSNs for the start of the redo phase and the end of the undo phase.*

The redo phase starts at `LSN = 3`, since this is the smallest `recLSN` in the dirty page table, and thus the earliest log record which could possibly have to be redone.

The undo phase also ends at `LSN = 3`, since in undoing transaction `T1` this is the earliest `LSN` we will encounter.

- *4. The set of log records that may cause pages to be rewritten during the redo phase.*

The set in question is `{3, 4, 5, 9}`.

To find this set, we first consider the set of *all* update log records: this is `{3, 4, 5, 6, 9}`. We then remove the record with `LSN = 6`, since the affected page exists in the DPT with `recLSN = 5`, which is lower than `6`.

- *5. The set of log records undone during the undo phase.*

The complete set of log records undone during the undo phase is `{9, 4, 3}` (in that order).

Here's why: Starting with the `lastLSN`s for `T1` (given by the transaction table), the undo phase traces log records backwards in time via the `prevLSN`s and `undoNextLSN`s of update records and CLRs.

- *6. The contents of the log after the recovery procedure completes.*

```
LSN | prevLSN  | Xact_ID | Type       | pageId   | undoNextLSN
----+----------+---------+------------+----------+------------
1   | -        | -       | begin CKPT | -        | -
2   | -        | -       | end CKPT   | -        | -
3   | NULL     | T1      | update     | P2       | -
4   | 3        | T1      | update     | P1       | -
5   | NULL     | T2      | update     | P5       | -
6   | NULL     | T3      | update     | P5       | -
7   | 6        | T3      | commit     | -        | -
8   | 5        | T2      | commit     | -        | -
9   | 4        | T1      | update     | P5       | -
10  | 7        | T3      | end        | -        | -
---------------------CRASH AND RESTART----------------------
11  | 8        | T2      | end        | -        | -
12  | 9        | T1      | CLR        | P5       | 4
13  | 12       | T1      | CLR        | P1       | 3
14  | 13       | T1      | CLR        | P2       | NULL
15  | 14       | T1      | end        | -        | -
```

Notice the `end` record for `T2`, which was written during the analysis phase.

---

5 More ARIES

- *1. Map the placeholders A-D to the correct LSNs.*

A = 5, since this CLR is for the update record with LSN = 6.

B = 4, since this CLR is for the update record with LSN = 8.

C = D = NULL.

- *2. What are the states of the transaction and dirty page tables after the analysis phase?*

```
/----------------------------\
|      Transaction Table     |
|----------------------------|
| xact id | status | last_LSN |
|---------+--------+----------|
| T1      | U      | 15       |
| T2      | U      | 16       |
\----------------------------/

/--------------------\
|  Dirty Page Table  |
|--------------------|
| page id | rec_LSN  |
|---------|----------|
| P3      | 3        |
| P1      | 4        |
| P2      | 6        |
| P4      | 8        |
\--------------------/
```

T3 is not in the transaction table after the analysis phase, since there exists an end record for it. T1 and T2 are in the transaction table since, and their lastLSNs are the last CLR corresponding to their respective abort recoveries; their statuses are set to U since they have not committed.

The dirty page table contains all four pages and their earliest associated LSNs after the last checkpoint.

- *3. Show the contents of the log after the crash recovery completes.*

```
LSN | PREV_LSN | XACT_ID | TYPE       | PAGE_ID | UNDONEXTLSN | toUndo after step
----+----------+---------+------------+---------+-------------+
1   | -        | -       | begin CKPT | -       | -           |
2   | -        | -       | end CKPT   | -       | -           |
3   | NULL     | T1      | update     | P3      | -           |
4   | 3        | T1      | update     | P1      | -           |
5   | NULL     | T2      | update     | P1      | -           |
6   | 5        | T2      | update     | P2      | -           |
7   | NULL     | T3      | update     | P2      | -           |
8   | 4        | T1      | update     | P4      | -           |
9   | 7        | T3      | commit     | -       | -           |
10  | 8        | T1      | abort      | -       | -           |
11  | 6        | T2      | abort      | -       | -           |
12  | 11       | T2      | CLR        | P2      | 5           |
13  | 9        | T3      | end        | -       | -           |
14  | 10       | T1      | CLR        | P4      | 4           |
15  | 14       | T1      | CLR        | P1      | NULL        |
16  | 12       | T2      | CLR        | P1      | NULL        |
---------------------CRASH AND RESTART-----------------------|
17  | 16       | T2      | CLR        | P1      | 12          |
18  | 17       | T1      | CLR        | P1      | 14          |
19  | 18       | T1      | CLR        | P4      | 10          |
19  | 18       | T1      | CLR        | P1      | 11          |
20  | 19       | T1      | CLR        | -       | 6           |
21  | 20       | T1      | CLR        | -       | 8           |
22  | 21       | T1      | CLR        | P4      | 4           |
23  | 22       | T2      | CLR        | P2      | 5           |
24  | 23       | T2      | CLR        | P1      | NULL        |
25  | 24       | T1      | CLR        | P1      | 3           |
26  | 25       | T1      | CLR        | P3      | NULL        |
```

I'm pretty sure there is something wrong with my solution. Some tips or pointers would be much appreciated! I think there is something wrong with my understanding of how to backtrace CLRs.

- *4a and 4b.*

I unfortunately did not have time to finish these.