UNIVERSITETET I OSLO

More concurrency control

Jon Grov and Ragnar Normann

Exercises from last week - 1

- **Exercise 1**: Consider the schedule
  $s = r_1(x)\ r_2(x)\ w_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$
  As usual, the initial and final transactions are implicit
  The Reads-From relation, *RF(s)* =
  $\{ (t_0, x, t_1), (t_0, x, t_2), (t_0, y, t_1), (t_0, y, t_2), (t_1, x, t_\infty)\ (t_2, y, t_\infty) \}$
  Explain why *s* is not view serializable
- Answer: Both t_1 and t_2 read *x* and *y* from t_0
  Since both transactions write an object read by the other, the Read-From relation for any serial plan schedule must contain ether (t_1, x, t_2) or (t_2, y, t_1)
  It follows that s cannot be view serializable

Last week's exercises – 2

- **Exercise 2:** Show that the following history is final state serializable but not view serializable
  $s = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$
  Answer (final state serializability):
  $H[s](x) = H_s(w_2(x)) = f_{2x}(H_s(r_2(x))) = f_{2x}(H_s(w_0(x))) = f_{2x}(f_{0x}())$
  $H[s](y) = H_s(w_2(y)) = f_{2y}(H_s(r_2(x)), H_s(r_2(y)))$
  $= f_{2y}(H_s(w_0(x)), H_s(w_0(y))) = f_{2y}(f_{0x}(), f_{0y}())$
  Put $s' = t_2t_1 = r_2(x)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ r_1(x)\ r_1(y)\ c_1$
  $H[s'](x) = H_{s'}(w_2(x)) = f_{2x}(H_{s'}(r_2(x))) = f_{2x}(f_{0x}())$
  $H[s'](y) = H_{s'}(w_2(y)) = f_{2y}(H_{s'}(r_2(x)), H_{s'}(r_2(y))) = f_{2y}(f_{0x}(), f_{0y}())$
  This shows that *s* and *s'* final state equivalent

Last week's exercises – 3

- **Exercise 2 continued:** Show that the following history is final state serializable but not view serializable
  $s = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$
  Answer (view serializability):
  $RF(s) = \{ (t_0,x,t_2), (t_2,x,t_1), (t_0,y,t_1), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty) \}$
  Put $s' = t_1t_2 = r_1(x)\ r_1(y)\ c_1\ r_2(x)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$
  $RF(s'') = \{ (t_0,x,t_1), (t_0,y,t_1), (t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty) \}$
  Put $s'' = t_2t_1 = r_2(x)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ r_1(x)\ r_1(y)\ c_1$
  $RF(s'') = \{ (t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_1), (t_2,y,t_1), (t_2,x,t_\infty), (t_2,y,t_\infty) \}$
  This shows that *s* is neither view equivalent to *s'* nor to *s''*

Introduction for today

- We are still working on the optimal scheduler.
- So far, we saw:
  - Final-state serializability is the "outer bound" - if we allow all final state-serializable schedules, we achieve maximum concurrency.
  - But we do not care about maximum concurrency, we care about maximum performance!
  - We saw view-serializability, which is easier to test than final-state, but still too expensive to use in a real scheduler.

Conflict-based scheduling

- Remember: A schedule *s* is serializable if it is *equal* to some serial schedule *s'*.
- So far, both equivalence-criterions have had the undesirable property that to determine this for an arbitrary schedule *s* is NP-complete.
- Today, we shall introduce *conflict equivalence* and *conflict serializability*.
- We shall see that to test whether a given schedule is conflict serializable can be done in polynomial time.
- Thus, we have drawn an important border: The best scheduling-algorithm providing conflict serializability is the best, general serializable scheduling-algorithm – there should be no need to look further.

Conflict equivalence

- Two operations in a schedule are in ***conflict*** if they
  - belong to different transactions
  - access the same data object
  - at least one of them is a write operation
- The definition of a schedule states that two conflicting operations in a schedule *s* must be ordered in $<_s$
- Definition:

  Two schedules *s* and *s'* are ***conflict equivalent*** if
  - *op(s) = op(s')*
  - all pairs of conflicting operations in *op(s)* appear in the same order in $<_s$ and $<_{s'}$

Conflict serializability

- Definition:

  A schedule is ***conflict serializable*** if it is conflict equivalent to a serial schedule
- Example: Consider the schedule

  $s = r_1(x)\ w_1(x)\ r_2(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$

  The set of conflicts in *s* is: $\{ (w_1(x), r_2(x)), (r_1(y), w_2(y)) \}$

  We have $w_1(x) <_s r_2(x)$ and $r_1(y) <_s w_2(y)$.

  In both conflicts the conflicting operation of t_1 precedes the conflicting operation of t_2

  Hence *s* is conflict equivalent to the serial schedule

  $s' = r_1(x)\ w_1(x)\ r_1(y)\ c_1\ r_2(x)\ r_2(y)\ w_2(y)\ c_2$

  proving *s* to be conflict serializable

Conflict vs view serializability

- Theorem:

  Every conflict serializable schedule is view serializable

  Proof: Let *s* be a schedule which is conflict equivalent to the serial schedule *s'*

  Assume for contradiction that $RF(s) \neq RF(s')$

  Let (t_k,x,t_i) be in $RF(s)$ but not in $RF(s')$, i.e.
  $r_i(x)$ reads from $w_k(x)$ in *s* but from some other $w_n(x)$ in *s'*

  Since *s* and *s'* are conflict equivalent we have
  $w_k(x) <_s r_i(x) \wedge w_k(x) <_{s'} r_i(x) \wedge w_n(x) <_{s'} r_i(x) \wedge w_n(x) <_s r_i(x)$

  But $w_k(x)$ and $w_n(x)$ are also in conflict, and we see that we must have $w_n(x) <_s w_k(x) \wedge w_k(x) <_{s'} w_n(x)$, a contradiction

  Thus $RF(s) = RF(s')$ which shows *s* to be view serializable

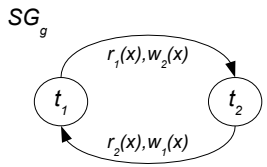
  QED

Conflict graphs (Serialization graphs)

- The conflict (or serialization) graph SG_s for a schedule *s* is a directed graph defined as follows:
  - Every transaction in $T(s)$ is represented as a node in SG_s
  - For all pairs t_1 and t_2 of transactions in $T(s)$ there is an edge from t_1 to t_2 in SG_s if and only if there is a conflict between two operations o_1,o_2, respectively belonging to t_1 and t_2, where o_1 is executed prior to o_2, i.e. $o_1 <_s o_2$
- The terms "serialization graph" and "conflict graph" are both used for SG_s, and both terms may be used at will
- A schedule *s* is conflict serializable if and only if SG_s is acyclic

Example

- Consider the schedule $s = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$
- Then each of t_1 and t_2 are represented as a node in SG_g
- We have two conflicts in *s*: $(r_1(x),w_2(x))$ and $(r_2(x),w_1(x))$
- The conflict graph is as follows:



- We observe that SG_g is cyclic
- Hence *s* is not conflict serializable

More about conflicts

- If t_i and t_k have a conflict in a schedule *s*, and t_i's operation comes first, *s* can only be equivalent to serial plans where t_i is performed prior to t_k
- Thus for any two transactions we must ensure that the execution order is the same in all conflicts between them
- However, operations not in conflict may be executed in an arbitrary order – they commute
- Note:
  Two operations are in conflict if and only if the final database state may be influenced by their execution order
- Pairs of write operations on one object never commute
- Pairs of read operations always commute

Testing conflict serializability

- To determine whether a schedule is conflict serializable we may construct its conflict graph and test it for cycles
- Constructing such a graph may be done in time proportional to the number of operations in the schedule, and testing for cycles in the graph is both in average and in worst case proportional to the square of the number of operations
- Thus we have a $O(n^2)$-algorithm, where n is the number of operations in the schedule, to decide whether the schedule is conflict serializable or not
- This shows that we finally have obtained an equivalence definition it is feasible to use
- The rest is pragmatics ....

Practical schedulers

- *So, with all this, it should now be simple:* update the serialization graph for each operation, check for cycles and block the operation or abort if a cycle is detected.
- And SG-testing has been implented in research prototypes.
- But...
  - Real transaction processing is more about engineering than mathematics.
  - Our SG-scheduler has a quite large run-time overhead, and restricting concurrency (by introducing a more blocking or aborts) will often pay off.
  - This is especially true if the chance of blocking or aborts is relatively low (i.e. different transaction racing for the same objects is rare)

Two-phase locking

- Two-phase locking (2PL) is very simple, ensures conflict-serializability and has very low runtime overhead.
- Transactions must obtain a lock before reading or writing an object.
- We have read-locks and write-locks:
  - Read-locks can be shared
  - Write locks are exclusive
- A very important property: As soon as a transaction has released its lock on object *o*, it may not acquire new locks on **any** object.
- A conflict-serializable schedule which cannot be produces by a 2PL-scheduler:
  $s = r_2(x)\ r_1(x)\ r_1(y)\ w_1(x)\ r_2(y)\ c_1\ c_2$

Example

$r_1(x)\ r_2(y)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ c_1\ c_2$

Scheduler:

```
request_read(t1,x)
read_lock(t1,x) → r1(x)
request_read(t2,y)
read_lock(t2,y) → r2(y)
request_write(t1,x)
write_lock(t1,x)→ w1(x)
request_read(t2,x)
request_read(t1,y)
read_lock(t1,y)→ r1(y)
unlock_all(t1)
read_lock(t1,x)→ r2(x)
request_write(t2,y)
write_lock(t2,y)→ w2(y)
unlock_all(t2)
```

t_2 is blocked since t_1's write lock is exclusive

t_1 is finished and releases all locks

$r_1(x)\ r_2(y)\ w_1(x)\ r_1(y)\ r_2(x)\ w_2(y)\ c_1\ c_2$

Challenges with 2PL

- Since the end of the 1970-ies, 2PL (in different variants) has been the predominant scheduling-protocol for transaction processing systems.
- But there is plenty of motivation to look for alternatives:
  - 2PL is relatively strict
  - It is not so well suited for distributed systems where remote locks require message sending
  - Deadlocks may be a problem

Non-locking schedulers

- So far, we have seen how *locks* may be used to block transactions preventively.
- We shall now see three different approaches:
  - Precommit-validation, aka *optimistic concurrency control*
  - Timestamp ordering
  - Multi-version concurrency control

Timestamp-ordering

- When initialized, all transactions are assigned a unique, monotoneusly growing timestamp. Any pair of conflicting operations must be executed in timestamp order – otherwise, one of the two transactions in conflict must abort.
- Example:
  $s = ts(t_m):=1, ts(t_n):=2, r_m(x), r_n(x), w_m(x)$
- Here, we must either abort t_m or t_n, $r_n(x) < w_m(x)$ in s while $ts(t_m) < ts(t_n)$
- Note that t_n may commit before t_m's write to x. In this case, we have no choice and must abort t_m.

Timestamp-ordering - 2

- Assuming we have a reasonable way to assign timestamps, the runtime-overhead of this approach has much lower runtime-overhead than 2PL in a distributed system, and this may pay well of if the chance of conflicts (and aborts) is low.
- The main disadvantage is that aborts are regarded as expensive compared to blocking, and “pure” timestamp ordering is rarely (if ever) seen in practice.

Optimistic concurrency control

- With optimistic concurrency control, transactions are executed in three phases:
  - *Read phase*: All read-operations are executed. Updates are buffered and kept invisble to other transactions.
  - *Validation phase:* When all operations have been tentatively executed and the transaction requests commit, the scheduler validates the execution. A transaction t_m can commit if we have that for every transaction t_n where $RS(t_n) \cap WS(t_m) \neq \emptyset$
  - *Write phase:* All tentative updates are made pemanent
- As with timestamp ordering, it is uncommon to see this particular algorithm in production systems.
- Nevertheless, the basic idea of deferring validation until transaction commit is well known and important.

Non-serializability

- The attractiveness of serializability is full transparency:
  - As long as we ensure that all schedules generated by our scheduler are serializable, the application developer should never need to worry about inconsistency due to concurrent execution.
- But as we have seen, this approach has to be conservative: Every read is assumed to affect every write.
- First: This approach is useless for interactive transactions, e.g. travel bookings. These are commonly implemented as a sequence of smaller, independent transactions – this topic will be discussed later.
- Second: Many applications and environments have performance requirements which makes fully transparent concurrency control impossible.
- But we still want some concurrency control. The problem is that the developer should be allowed to say: *We don't care about the ordering of these operations, even if they conflict.*

Concurrency phenomena

- In an attempt to systematize non-serializable scheduling, the SQL-standard defines the following *concurrency phenomena*:
  - **Dirty read:** If a transaction reads data written by a concurrent, non-committed transaction
  - **Nonrepeatable read**: If a transaction reads the same data twice, and the values are changed meanwhile by some other, now committed transaction
  - **Phantom read:** If a transaction execute a query with predicate *P* twice, and the number of rows returned is changed

Isolation levels

	Dirty read	Nonrepeatable read	Phantom read
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

Exercise

1. Suggest an isolation level for the following transactions:

   A) BEGIN:
   nuts = 'SELECT * FROM components WHERE type = 'red'
   FOREACH *c* IN *nuts*:
   UPDATE components SET stock = stock + 1;
   COMMIT;

   B) BEGIN;
   totalweight = 'SELECT SUM(weight) FROM components;'
   'UPDATE componets SET fraction = weight/*totalweight*;'
   COMMIT;

2. Oracle's default isolation level is *READ COMMITTED*. Oracle provides a non-standard statement *SELECT ... FOR UPDATE*. What could this be used for?

Multiple versions

- So far, we have assumed that on each update, the previous object value is overwritten.
- But in many circumstances, disk and memory is a minor problem.
- Consequently, modern database systems ususally keep previous versions (for a while)
- The main purpose of this is increased concurrency:
  - **Without** multiversioning: $s = r_1(x)\ r_1(y)\ w_1(x)\ r_2(x)\ r_2(y)\ w_1(y)$
  - **With** multiversioning: $s' = r_1(x)\ r_1(y)\ w_1(x_{new})\ r_2(x_{old})\ r_2(y_{old})\ w_1(y_{new})$
- Note that *s* is not serializable, but *s'* is final-state equivalent with the serial schedule $s'' = r_2(x)\ r_2(y)\ r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

Multiversion-schedules

- Multiversion-schedules is an extension of our previous schedule-conept as operations now execute on object *versions*.
- A new version is created each time a transaction updates an object, and we identify a given version by the id of the creating transaction.
- We still assume a (fictitious) initializing transaction t_0 creating an initial version of all objects accessed in the schedule.
- Example:
  s = $r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ r_2(x_1)\ r_2(y_0)\ w_1(y_1)$
- In this model, an update always creates a new version, while a read may access any previously created version.

Multiversjon-scheduling

- A multiversion-schedule accepts an ordinary plan as input and produces a multiversion-schedule as output.i
- Thus, the scheduler now has two tasks:
  - Decide an order for executing operations
  - Apply a given *version-function* to decide which version a read-operation reads from
- A traditional scheduler can be regarded as a special case where the version-function always chooses the most recently created version.
- In this context, traditional (non-versioned) schedules are denoted *monoversion-schedules.*
- The acronym *MVCC* (MultiVersion Concurrency Control) is commonly used to describe transaction processing systems providing some kind of multiversion-support.

Correctness of multiversion-schedules

- Requiring equality with a *serial* multiversion schedule does not really make sense, since the read-operations in principle can read any previous version.
- Thus, final-state seriliazability for multiversion schedules are defined as follows:

  A multiversion-schedule *m* is final-state serializable if and only if there exists some serial *monversion* plan *m'* where *m* and *m* contains the same operations and are final-state equivalent.

The optimal MVCC-scheduler

- The previous definition of *conflict* is based on commutativity, i.e. the property that reordering a pair of conflicting operations may change the final state.
- In a multiversion-setting, we must know the version-function to decide whether a pair of operations are commutable or not.
- Thus, we cannot apply conflict-graph testing to check whether a multiversion-schedule is serializable.
- On the other hand, we observe that view-serializability is independent of the version-function: The reads-from relation is well-defined for any multiversion-schedule independent of the version-function.
- But since view-serializability cannot be decided in polynomial time, there is no (theoretically) optimal scheduler to provide complete multiversion-serializability.

MVCC in practice

- Multiversion-scheduling is widely used in practice, and the two most important variants are:
  - Read-only snapshot (ROS)
  - Snapshot isolation (RI)
- Both scheduling-strategies are based on a *commit*-snapshot:
  - A version x_i, written by transaction t_i, is included in the commit-snapshot for a transaction t_j if and only if t_i was the last transaction to update *x* among all committed transactions at t_j's initialization.
  - In practice, this is implemented by timestamping transactions at initialization- and commit-time.

Read-only snapshot

- ROS is a *hybrid algorithm*, and the application should declare whether a transaction is read-only in the request.
- A read-only transaction reads from its commit-snapshot: No locking or validation is required.
- Update-transactions are handled by some conventional scheduler, e.g. 2PL.
- Example:
- $m = w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_1)\ w_2(x_2)\ c_2\ r_3(y_1)\ \ w_4(x_4)\ w_4(y_4)\ c_4\ r_3(x_2)\ c_3$
- Note that x_2 is part of the commit-snapshot of t_3, while x_1 and x_4 are not.
- $LF(m) = \{ (t_1, y, t_3), (t_1, x, t_2), (t_2, x, t_3), (t_4, x, t_\infty), (t_4, y, t_\infty) \}$
- Consequently, *m* is view equivalent with the serial monoversion-schedule $s = t_1 t_2 t_3 t_4$

Snapshot isolation

- Snapshot Isolation (SI) is a “pragmatic” mutliversion scheduling algorithm, and it is used by Oracle, MS SQL Server, MySQL and PostgreSQL.
- With SI, *all* transactions read from their respective commit-snapshot.
- But we require the write-sets of any two concurrent transactions to be disjoint.
- SI is Oracle's implementation of the “serializable” isolation level, but it is not really serializable.
- Oracle's standard isolation level is *read committed*, which is snapshot isolation where all read-operations read from the *read committed*-snapshot, defined as follows:

  A version x_i, written by transaction t_i, is part of the *read committed*-snapshot for a read-operation $r(x)$ if and only if t_i is the most recently committed transaction among all updaters of x at the time $r(x)$ is executed.