Transaction Systems
Exercise Session 5

Andrey Gubichev

December 2, 2013
Homework: Task 1.1, Transaction Isolation

Execute the following two transactions and write down the results of the SELECT COUNT() statements:

<table>
<thead>
<tr>
<th>Window 1</th>
<th>Window 2</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0     BEGIN TRANSACTION;</td>
<td>BEGIN TRANSACTION;</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>SELECT COUNT(*) FROM City;</td>
<td>3427</td>
</tr>
<tr>
<td></td>
<td>SELECT COUNT(*) FROM Person</td>
<td>10000</td>
</tr>
<tr>
<td>2     INSERT INTO City VALUES (...)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3     SELECT COUNT(*) FROM City;</td>
<td>3428</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SELECT COUNT(*) FROM City;</td>
<td>3427</td>
</tr>
<tr>
<td>5     INSERT INTO Person VALUES(...)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>6     SELECT COUNT(*) FROM Person</td>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SELECT COUNT(*) FROM Person;</td>
<td>10000</td>
</tr>
<tr>
<td>8     COMMIT;</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>SELECT COUNT(*) FROM City;</td>
<td>3428</td>
</tr>
</tbody>
</table>
Homework: Task 1.2, Isolation Level

Execute the following two transactions and write down the results of the `SELECT COUNT()` statements:

<table>
<thead>
<tr>
<th>Window 1</th>
<th>Window 2</th>
<th>Count</th>
</tr>
</thead>
</table>
| 0 BEGIN TRANSACTION;      | **BEGIN TRANSACTION;**
|                           | **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;**                      | —       |
| 1                         | **SELECT COUNT(*) FROM City;**                                          | 3427    |
|                           | **SELECT COUNT(*) FROM Person**                                         | 10000   |
| 2 INSERT INTO City VALUES (....) |                                                                           | —       |
| 3 SELECT COUNT(*) FROM City; |                                                                           | 3428    |
| 4                         | **SELECT COUNT(*) FROM City;**                                          | 3427    |
| 5 INSERT INTO Person VALUES(..) |                                                                           | —       |
| 6 SELECT COUNT(*) FROM Person |                                                                           | 10001   |
| 7                         | **SELECT COUNT(*) FROM Person**                                         | 10000   |
| 8 COMMIT;                 |                                                                           | —       |
| 9                         | **SELECT COUNT(*) FROM City;**                                          | 3427    |
| 10                        | COMMIT;                                                                  | —       |
| 11                        | **SELECT COUNT(*) FROM City;**                                          | 3428    |
Ordered sharing of locks

- History $s = r_1(x)w_2(x)r_3(y)c_3 w_1(y)c_1 w_2(z)c_2$
- is in CSR, but not accepted by 2PL (i.e., not in Gen(2PL))
- Let’s relax the restrictions on locking
Ordered sharing of locks

- Two locks on the same data item can be held simultaneously if the lock operations and the corresponding data operations are executed in the same order.

- \( pl_i(x) \rightarrow ql_j(x), \ i \neq j \) means \( pl_i(x) <_s ql_j(x) \) and \( p_i(x) <_s q_j(x) \)
Ordered sharing in lock tables

<table>
<thead>
<tr>
<th>(LT_1)</th>
<th>(rl_i(x))</th>
<th>(wl_i(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(rl_j(x))</td>
<td>(\sqrt{\phantom{a}})</td>
<td>–</td>
</tr>
</tbody>
</table>
### Ordered sharing in lock tables

<table>
<thead>
<tr>
<th>$LT_1$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$-$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_2$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_3$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$-$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$\rightarrow$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_4$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$-$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$-$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_5$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$\rightarrow$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_6$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$-$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_7$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$-$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_8$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>
Example

- History \( s = w_1(x) r_2(x) r_3(y) c_3 w_1(y) c_1 w_2(z) c_2 \)
- Lock table \( LT_2 \)

<table>
<thead>
<tr>
<th>( LT_2 )</th>
<th>( rl_i(x) )</th>
<th>( wl_i(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rl_j(x) )</td>
<td>( \sqrt{\quad} )</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>( wl_j(x) )</td>
<td>( _ )</td>
<td>( _ )</td>
</tr>
</tbody>
</table>
Lock acquisition and release

- Operations are executed in the order that lock have been acquired
- But this is not enough for conflict serializability
- $w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1$
Lock acquisition and release

- Operations are executed in the order that lock have been acquired
- But this is not enough for conflict serializability
- $w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1$
- Once $T_2$ acquires lock on $x$, $T_1$ should not get any conflicting lock with $T_2$
- After $T_2$ commits, $T_1$ can again acquire the write lock: conflict!
- Here, $T_2$ is on hold (order dependent on $T_1$, $T_1$ has not released any lock yet)
- One more rule: if transaction is on hold, it can not release any locks
O2PL

- O2PL is 2PL with the table LT8 and two additional rules (acquisition, release)

<table>
<thead>
<tr>
<th>LT_8</th>
<th>(rl_i(x))</th>
<th>(wl_i(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(rl_j(x))</td>
<td>(\sqrt{\phantom{\text{}}})</td>
<td>→</td>
</tr>
<tr>
<td>(wl_j(x))</td>
<td>→</td>
<td>→</td>
</tr>
</tbody>
</table>
Homework: Task 3

\[ s_1 = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1 \]
\[ w_1(x)w_1(x)rl_2(x)r_2(x)... \]
Homework: Task 3

- \( s_1 = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1 \)
- \( w_1(x)w_1(x)rl_2(x)r_2(x)rl_3(y)r_3(y)ru_3(y)c_3wl_1(y)w_1(y) \\
  wu_1(y)wu_1(x)ru_2(x)c_2c_1 \)
- \( T_2 \) is on hold
Homework: Task 3

- $s_1 = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1$
- $wl_1(x)w_1(x)rl_2(x)r_2(x)rl_3(y)r_3(y)ru_3(y)c_3wl_1(y)w_1(y)$
  $wu_1(y)wu_1(x)ru_2(x)c_2c_1$
- Or: ...$wu_1(y)wu_1(x)c_1ru_2(x)c_2$. This would be the only option if the protocol was strict
(Basic) Timestamp ordering

- Non-locking algorithm (unlike previous ones)
- Transactions get timestamps $ts$ (time of beginning, number)
- TO rule: if $p_i(x)$ is in conflict with $q_j(x)$, then $p_i(x)$ is executed before $q_j(x)$ iff $ts(T_i) < ts(T_j)$
The scheduler executes the operations right away, except the operation is too late.

Operation $p_i(x)$ is too late, if the conflicting operation $q_j(x)$ of a later transaction has already been executed:

- $p_i(x)$ and $q_j(x)$ are in conflict
- $q_j(x)$ has been executed
- but $ts(T_j) > ts(T_i)$
- $p_i(x)$ is too late!
(Basic) Timestamp ordering

For every \( x \) record

- \( \text{max-r-scheduled}(x) \) – max timestamp of the read operation sent to data manager so far
- \( \text{max-w-scheduled}(x) \) – max timestamp of the write operation sent to data manager so far
Example

- $ts(T_1) = 1$, $ts(T_2) = 2$, $ts(T_3) = 3$
- $s = r_1(x)w_2(x)r_3(y)w_2(y) \cdot w_3(z)c_3r_1(z)$
Example

- $ts(T_1) = 1$, $ts(T_2) = 2$, $ts(T_3) = 3$
- $s = r_1(x)w_2(x)r_3(y)w_2(y)a_2w_3(z)c_3r_1(z)a_1$
Homework: Task 4

- $s_1 = r_1(x)r_2(x)c_1c_2$ belongs to both classes
- $s_2 = r_1(x)w_1(y)r_2(x)w_2(y)r_1(z)c_1c_2 \in \text{Gen(BTO)}, \notin \text{Gen(2PL)}$
- $s_3 = r_1(x)w_1(x)w_2(y)r_1(y)c_1c_2 \in \text{Gen(2PL)}, \notin \text{Gen(BTO)}$
Plan for today

- Serialization Graph Testing (SGT)
- Optimistic protocols: BOCC, FOCC
- Multiversion Concurrency Control
SGT: Overview

- Goal: keep the CG acyclic
- Maintain a graph in which nodes and edges are added dynamically
A new operation $p_i(x)$ arrives:

- create a node for $T_i$ if it does not exist
- insert edges $(T_j, T_i)$ for all transactions in conflict
  - There is a cycle. Reject $p_i(x)$, abort $T_i$
  - No cycle. Add the edge permanently
Graph tends to grow over time

Can we remove information about old transactions (committed long time ago)?
SGT: Discussion

- Graph tends to grow over time
- Can we remove information about old transactions (committed long time ago)?

\[ r_{k+1}(x) w_1(x) w_1(y_1) c_1 w_2(x) w_2(y_2) c_2 \ldots w_k(x) w_k(y_k) c_k w_{k+1}(z) \ldots \]

- Need to check whether \( z \in \{ x, y_1, \ldots, y_k \} \)
SGT: Discussion

- Graph tends to grow over time
- Can we remove information about old transactions (committed long time ago)?

\[
r_{k+1}(x) w_1(x) w_1(y_1) c_1 w_2(x) w_2(y_2) c_2 \ldots w_k(x) w_k(y_k) c_k w_{k+1}(z) \ldots
\]

  all of \( t_1 \)

  all of \( t_2 \)

  all of \( t_k \)

- Need to check whether \( z \in \{x, y_1, \ldots, y_k\} \)
- if \( T_i \) is finished and is a source (no incoming edges), it can be safely removed (incoming edges are never added afterwards)
Optimistic protocols: main ideas

- Assume conflicts are rare
- Let the arriving operations simply pass, but sometimes check whether the schedule is serializable
- Checking *(validation)* is part of the transaction
- Transaction has three phases:
  - Read phase: all writes are applied to the private workspace (not to the database). No one else can see them
  - Validate phase: ready to commit. Was the execution OK (CSR?)
  - Write phase: workspace is transferred into the database
- Atomic *Val-Write* phase
Validation approaches

▶ backward-oriented (BOCC): a transaction under validation executes a conflict test against the already committed transactions
▶ forward-oriented (FOCC): T is validated against the transactions that run in parallel, but are still in their read phase
▶ Both guarantee that the CG is acyclic
$T_j$ is under validation. For every committed $T_i$ check:

- $T_i$ ended before $T_j$ started.
- $RS(T_j) \cap WS(T_i) = \emptyset$ and $Val-Write(T_i)$ ended before $Val-Write(T_j)$ ($T_j$ does not read from $T_i$)

If not: abort $T_j$
BOCC Example

read phase

\[ t_1 \]
\[ r_1(x) \quad r_1(y) \quad \text{val.} \quad w_1(x) \]

write phase

\[ t_2 \]
\[ r_2(y) \quad r_2(z) \quad \text{val.} \quad w_2(z) \]

\[ t_3 \]
\[ r_3(x) \quad r_3(y) \quad \text{val.} \quad \text{abort} \]

\[ t_4 \]
\[ r_4(x) \quad \text{val.} \quad w_4(x) \]
$T_j$ is under validation at the time $n$. For every running $T_i$, $RS^n(T_i)$ is the read set of $T_i$ at the moment $n$. Check:

- For all $T_i$ reading at the current moment $n$:
  \[ WS(T_j) \cap RS^n(T_i) = \emptyset \]

If not:

- abort $T_i$
- abort $T_j$ for which the intersection is non-empty
- wait with validation
FOCC Example

read phase

\[ r_1(x) \quad r_1(y) \quad \text{val.} \quad w_1(x) \]

write phase

\[ r_2(y) \quad r_2(z) \quad \text{val.} \quad w_2(z) \]

abort

\[ r_3(z) \]

\[ r_4(x) \quad r_4(y) \quad \text{val.} \quad w_4(y) \]

\[ r_5(x) \quad r_5(y) \]
Multiversion concurrency control: ideas

- Write operations are no longer "in place". Each write operation creates a version of the data item
- Old values are always accessible
- Scheduler has to decide which version to read
A function $h$:

- $h(w_i(x)) = w_i(x)$, $w_i(x)$ writes $x_i$
- $h(r_i(x)) = w_j(x)$, for some $w_j(x) < s r_i(x)$, $r_i$ reads $x_j$

Multiversion history: history + version function
Example

\[ m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_0)w_1(z_1)c_1c_2 \]

- \( r_1 \) reads the initial version of \( x \): \( x_0 \) etc
- \( w_2 \) writes a version of \( y \) (\( y_2 \)), but \( r_1 \) reads the initial version of \( y \) (\( y_0 \))
Multi- vs Monoversion histories

- Generally, version function: \( r \) reads the item written by some preceding \( w \)
  - \( m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_0)w_1(z_1)c_1c_2 \)

- Monoversion history (everything before now): \( r \) reads the item written by the last preceding \( w \)
  - \( m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_2)w_1(z_1)c_1c_2 \)
Multiversion serializability

- Versions are invisible to user
- Correct multiversion schedule should be equivalent to the monoversion schedule
Reads-from relationship

- **Motivation**
  - \( s = w_0(x)c_0w_1(x)c_1r_2(x)w_2(y)c_2 \)
  - \( m = w_0(x_0)c_0w_1(x_1)c_1r_2(x_0)w_2(y_2)c_2 \)
  - Yield different values of \( y \) (under Herbrand semantics)
  - \( RF(m) = \{(t_i, x, t_j) | r_j(x_i) \in op(m)\} \)
  - View equivalence for \( m, m' \): same set of transactions, \( RF(m) = RF(m') \)
Example

- \( w_0(x_0)w_0(y_0)c_0 r_3(x_0)w_3(x_3)c_3 w_1(x_1)c_1 r_2(x_1)w_2(y_2)c_2 \)
- \( w_0(x_0)w_0(y_0)c_0 w_1(x_1)c_1 r_2(x_1)r_3(x_0)w_2(y_2)w_3(y_3)c_3 c_2 \)
MV view serializability

- How to define view serializability? ”View equivalent to a serial multiversion schedule”??
- No: serial *multiversion* schedule does not have to be compatible with serial *monoversion* schedule:
  - $m = w_0(x_0)w_0(y_0)c_0 r_1(x_0)r_1(y_0)w_1(x_1)w_1(y_1)c_1 r_2(x_0)r_2(y_1)c_2$
  - $s = w_0(x_0)w_0(y_0)c_0 r_1(x)r_1(y)w_1(x)w_1(y)c_1 r_2(x)r_2(y)c_2$
  - both serial, but different reads-from relation
- MV view serializable schedule $m$: there exists a serial *monoversion* schedule, view equivalent to $m$. 
How to test MVSR?

▶ There is a construct called Multiversion Serialization Graph
▶ If it is acyclic, the schedule is in MVS
▶ Important: NP-hard
▶ See lectures + homework
Multiversion Conflicts

- Previously, the conflicts were of the form
  - rw
  - wr
  - ww

- For multiversion schedule, only rw is a conflict
  - ww is no conflict: everyone writes his own version
  - wr: commuting the pair is no problem (restricts the version choices for read that still render the correct schedule)
  - rw: commuting the pair creates a problem (adds more choices for reading)
Multiversion Conflicts

- Previously, the conflicts were of the form
  - rw
  - wr
  - ww

- For multiversion schedule, only rw is a conflict

- Why?
  - ww is no conflict: everyone writes his own version
  - wr: commuting the pair is no problem (restricts the version choices for read that still render the correct schedule)
  - rw: commuting the pair creates a problem (adds more choices for reading)
Multiversion Conflict Serializability

- MV Conflict reducibility
- MCSR: there is a serial monoversion history with the same ordering of multiversion conflicts
- Multiversion conflict graph: edges of the form $r_i(x_j) < w_k(x_k)$ for the same data item $x$
- MCSR: Multiversion conflict graph is acyclic
Homework (1/2)

- For the following schedule:
  \[ r_1(x) r_2(x) r_1(y) r_3(x) w_1(x) w_1(y) c_1 r_2(y) r_3(z) w_3(z) c_3 r_2(z) c_2 \]
  show the output of BOCC and FOCC. Remember that write steps are actually performed on private workspaces, the commit requests initiate the validation, and the write steps are performed after successful validation.

- For the following histories test if they are MVSR or MCSR.
  - \[ w_0(x_0) w_0(y_0) w_0(z_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) c_1 r_2(x_1) w_2(y_2) w_2(z_2) c_2 \]
  - \[ w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_3(x_1) w_3(x_3) r_2(x_1) c_3 w_2(y_2) c_2 \]
  - \[ w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_2(x_1) w_2(y_2) c_2 r_3(y_0) w_3(x_3) c_3 \]
For the schedule

\[ m = w_0(x_0)w_0(y_0)c_0r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)w_1(y_1)w_3(y_3) \]

test whether there exists an order \(<<\) such that
\( \text{MVSG}(m, <\leq) \) is acyclic. If there is an acyclic graph, find an appropriate version function for a final transaction \( t_\infty \) such that the graph remains acyclic.
Info

- Exercises due: 9 AM, December 9, 2013
- Submit to andrey.gubichev@in.tum.de