CS4400: Transactions - implementation
Prof. Segre
11/15/2016
Minute paper

What is the importance of conflict serializability?
1 more example of testing conflict serializability

\[ r_1(A), r_3(A), w_3(A), r_2(B), w_1(B), w_2(B) \]
1 more example of testing conflict serializability

\[ r_1(A), r_3(A), w_3(A), r_2(B), w_1(B), w_2(B) \]

Not acyclic $\Rightarrow$ schedule is not conflict serializable $\Rightarrow$ schedule is not serializable
How does the database ensure serializability?

• *Scheduler* or *concurrency control* schedules the actions of transactions so that they are serializable

• Some concurrency control mechanisms?
  • Locking
  • Read-write sets
  • Multi-versioning
Simple locking

• Some rules
  • Every record has a single lock
  • Transaction must acquire the lock for a record before reading or writing it
  • Transaction must eventually release its acquired locks

• Do these rules ensure serializability?
Simple locking

• Some rules
  • Every record has a single lock
  • Transaction must acquire the lock for a record before reading or writing it
  • Transaction must eventually release its acquired locks

• One more rule
  • A transaction must release locks only after all locks it will use have been acquired

• Do these rules ensure serializability?
Another problem: recoverability

$L_1(A)$
$W_1(A)$
$U_1(A)$

$L_2(A)$
$R_2(A)$
$W_2(A)$
$U_2(A)$

rollback$_1$

commit$_2$

2PL does not guarantee *recoverability*: that we can restore the database to a previous state on rollback
Strict 2PL

• 2PL + all unlocks must happen in the commit/rollback

• Ensures that if anyone else sees T’s effects, then T was committed

\[
\begin{align*}
L_1(A) \\
W_1(A) \\
U_1(A), \text{ rollback}_1 \\
\quad \begin{align*}
L_2(A) \\
R_2(A) \\
W_2(A) \\
U_2(A), \text{ commit}
\end{align*}
\end{align*}
\]
Another problem: deadlocks

Example: One person transfers from account A to B and another person transfers from account B to A

$L_1(A)$                 \hspace{2cm} $L_2(B)$
Waiting to $L_1(B)$     Waiting to $L_2(A)$
...

Solutions for deadlock?
Performance

• Locking an object serializes access to it, decreasing performance of the overall system

• What are performance tradeoffs of locks OR ways to improve the performance of locks?
Performance tip: consider commutativity, part 1

- **Observation**: Reads commute with each other (we’ve already mentioned this fact while discussing conflict serializability)

- improve read performance by having two lock types: Shared (S) and Exclusive (X)
  - Shared: required for reading, many transactions may hold it
  - Exclusive: required for writing or reading, only one transaction may hold it and no transaction can hold a Shared

<table>
<thead>
<tr>
<th>Requested lock</th>
<th>Lock held by another transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>yes</td>
</tr>
<tr>
<td>X</td>
<td>no</td>
</tr>
</tbody>
</table>
Performance tip: consider commutativity, part 2

• **Observation**: Certain *write* operations are commutative: i.e., although they involve writes, they do not conflict.

• Example: increments
  • Increments commute with each other
  • But, reads and writes do not commute with increments!
  • Another lock type: *increment lock*

<table>
<thead>
<tr>
<th>Requested lock</th>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>I</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Performance tip: reduce the duration a lock is held

L₁(A)
R₁(A)
wait for L₂(A)

read message going to client
Client working
Write message going to database

W₁(A)

U₁(A), commit

L₂(A)

• T2 waits for a significant amount of time for round-trip messages between client and database
• If possible, put a transaction inside a stored procedure. Then there are no long-latency network messages between lock and unlock.
  • When is it not possible? Example: when transaction depends on human interaction at the client or complex code that cannot run in the database
Performance tradeoff: lock granularity

- **Fine grain** locking
  - Many locks on smaller objects (e.g., records)
  - + high concurrency
  - - high lock management cost

- **Coarse grain** locking
  - Fewer locks on larger objects (e.g., pages, tables)
  - + low lock management cost
  - - low concurrency
Lock granularity in practice

• How do you allow locking tables for some operations and records for others?
• One solution: “intention locks”

<table>
<thead>
<tr>
<th>Requested lock</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS held by another transaction</td>
<td>IS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IX held by another transaction</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>S held by another transaction</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>X held by another transaction</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

IS=intent to shared lock
IX=intent to excl lock

Also used for other locking other hierarchical structures, like B-trees

e.g. 3 transactions
• update A where id=c;
• update A where id=d;
• select * from A;
Performance summary

• Consider **commutativity** and use special locks for these operations

• Consider **lock granularity**

• Aside: lock compatibility tables are useful for specifying lock behavior
ACID and locks

- What components of ACID do locks (using strict 2PL) alone provide?
Summary

- How to test for serializability
- How to implement transactions with locks
- Performance of locks

- Next time:
  - How to maintain Atomicity
  - scalable transactional systems