CSE 486/586 Distributed Systems
Concurrency Control --- 2

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap
• Question: How to support transactions (with locks)?
  – Multiple transactions share data.
• Complete serialization is correct, but performance and abort are two issues.
• Executing transactions concurrently for performance
  – Problem: Not all current executions produce a correct outcome

Recap: Conflicting Operations
• Two operations are said to be in conflict, if their combined effect depends on the order they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, not on different variables.

<table>
<thead>
<tr>
<th>Operations of different</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a read and a write operation depends on the order of their execution</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of write operations depends on the order of their execution</td>
</tr>
</tbody>
</table>

Recap: Serial Equivalence
• An interleaving of the operations of 2 or more transactions is said to be serially equivalent if the combined effect is the same as if these transactions had been performed sequentially (in some order).

Recap: Serial Equivalence
• How to provide serial equivalence with conflicting operations?
  – Execute all pairs of conflicting operations in the same order for all objects

Recap: Serial Equivalence
• How to provide serial equivalence with conflicting operations?
  – Execute all pairs of conflicting operations in the same order for all objects

Pairs of Conflicting Operations
Handling Abort()

- What can go wrong?

<table>
<thead>
<tr>
<th>Transaction T₁</th>
<th>Transaction W₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100);</td>
<td>allBranch.branchTotal();</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>$100</td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>total = a.getBalance();</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>$300</td>
</tr>
<tr>
<td></td>
<td>total = total + b.getBalance();</td>
</tr>
<tr>
<td></td>
<td>total = total + c.getBalance();</td>
</tr>
</tbody>
</table>

Strict Executions of Transactions

- Transactions should **delay both their read and write operations** on an object (until commit time)
  - Until all transactions that previously wrote that object have either committed or aborted
  - This way, we avoid making intermediate states visible before commit, just in case we need to abort.
  - This is called **strict executions**.

- Thus, correctness criteria for transactions:
  - Serial equivalence
  - Strict execution

Story Thus Far

- Question: How to support transactions (with locks)?
  - Multiple transactions share data.
- First strategy: Complete serialization
  - One transaction at a time with one big lock
  - Correct, but at the cost of performance
- How to improve performance?
  - Let’s see if we can concurrently execute transactions.
- Problem: Not all current executions produce a correct outcome
  - Serial equivalence & strict execution must be met.
- Now, how do we meet the requirements using locks?
  - Overall strategy: using more and more fine-grained locking
  - No silver bullet. Fine-grained locks have their own implications.

Using Exclusive Locks

- Exclusive Locks (Avoiding One Big Lock)

Using Exclusive Locks

- Two phase locking
  - To satisfy serial equivalence
  - First phase (growing phase): new locks are acquired
  - Second phase (shrinking phase): locks are only released
  - A transaction is not allowed to acquire any new lock, once it has released any one lock
- Strict two phase locking
  - To satisfy strict execution, i.e., to handle abort() & failures
  - Locks are only released at the end of the transaction, either at commit() or abort(), i.e., the second phase is only executed at commit() or abort().
Non-Exclusive Locks

<table>
<thead>
<tr>
<th>Non-exclusive lock compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>none</td>
</tr>
<tr>
<td>read</td>
</tr>
<tr>
<td>write</td>
</tr>
</tbody>
</table>

- A read lock is promoted to a write lock when the transaction needs write access to the same object.
- A read lock shared with other transactions’ read lock(s) cannot be promoted. Transaction waits for other read locks to be released.
- Cannot demote a write lock to read lock during transaction – violates the 2P principle.

2PL: a Problem

- What happens in the example below?

Example: Non-Exclusive Locks

Transaction T1
- OpenTransaction()
- balance = b.getBalance()
- b.setBalance = balance * 1.1

Transaction T2
- OpenTransaction()
- balance = b.getBalance()
- b.setBalance = balance * 1.1
- Commit

Deadlock Conditions

- Necessary conditions
  - Non-sharable resources (locked objects)
  - No lock preemption
  - Hold & wait or circular wait
Preventing Deadlocks

- Acquiring all locks at once
- Acquiring locks in a predefined order
- Not always practical:
  - Transactions might not know which locks they will need in the future
- One strategy: timeout
  - If we design each transaction to be short and fast, then we can abort() after some period of time.

Two-Version Locking

- Three types of locks: read lock, write lock, commit lock
  - Transaction cannot get a read or write lock if there is a commit lock
- At commit(),
  - Promote all the write locks of the transaction into commit locks
  - If any objects have outstanding read locks, transaction must wait until the transactions that set these locks have completed and locks are released
- Compare with read/write locks:
  - Read operations are delayed only while transactions are committed
  - Read operations of one transaction can cause a delay in the committing of other transactions

Two-Version Locking lock compatibility

<table>
<thead>
<tr>
<th>Lock already set</th>
<th>Lock requested</th>
<th>read</th>
<th>write</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>OK</td>
<td>WAIT</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>OK</td>
<td>WAIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>WAIT</td>
<td></td>
<td>WAIT</td>
<td></td>
</tr>
</tbody>
</table>

Extracting Even More Concurrency

- Allow writing tentative versions of objects
  - Letting other transactions read from the previously committed version
- Allow read and write locks to be set together by different transactions
  - Unlike non-exclusive locks
- Disallow commit if other uncompleted transactions have read the objects
  - These transactions must wait until the reading transactions have committed
- This allows for more concurrency than read-write locks
  - Writing transactions risk waiting or rejection when commit

Summary

- Strict Execution
  - Delaying both their read and write operations on an object until all transactions that previously wrote that object have either committed or aborted
- Strict execution with exclusive locks
  - Strict 2PL
- Increasing concurrency
  - Non-exclusive locks
  - Two-version locks
  - Etc.

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).