COMP163
Database Management Systems
December 4, 2008

Concurrency Control – Chapter 18
Recovery – Chapter 19
### 2PL Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1(Y)</td>
<td>s2(Z)</td>
</tr>
<tr>
<td>r1(Y)</td>
<td>r2(Z)</td>
</tr>
<tr>
<td>x1(X)</td>
<td></td>
</tr>
<tr>
<td>u1(Y)</td>
<td>x2(Y)</td>
</tr>
<tr>
<td>r1(X)</td>
<td>u2(Z)</td>
</tr>
<tr>
<td>w1(X)</td>
<td>r2(Y)</td>
</tr>
<tr>
<td>u1(X)</td>
<td>w2(Y)</td>
</tr>
<tr>
<td></td>
<td>u2(Y)</td>
</tr>
</tbody>
</table>

Both transactions obey 2PL. It is not possible to interleave them in a manner that results in a non-serializable schedule.
Basic 2PL

- Basic 2PL requires that no locks be requested after the first unlock

- Guarantees serializability
  - transactions that request operations that violate serializability are delayed while waiting on locks

- Reduces concurrency, since locks must be held until all needed locks have been acquired

- May cause deadlock
Conservative 2PL

- Conservative 2PL requires that all locks must be acquired at start of transaction.
- Prevents deadlock, since all locks are acquired as a block:
  - No transaction can be waiting on one lock while it holds another lock.
- Further restricts concurrency, since transaction must request strongest lock that might be needed.
Strict 2PL

- Strict 2PL requires that all locks must be held until end of transaction
- Deadlock is possible
- Guarantees strict schedules
- May require holding locks longer than necessary
- Most commonly used algorithm
Serializablility: Commutativity

- Two operations commute if, when executed in either order:
  - The values returned by both are the same and
  - The database is left in the same final state

- Two schedules are equivalent if one can be derived from the other by a series of simple interchanges of commutative operations

- A schedule is serializable if it is equivalent to a serial schedule
Serializability: Commutativity

- $S: r_1(x) \ w_2(z) \ w_1(y)$
  
is equivalent either serial schedules of $T_1$ and $T_2$
  
  $T_1, T_2: \ r_1(x) \ w_1(y) \ w_2(z)$
  $T_2, T_1: \ w_2(z) \ r_1(x) \ w_1(y)$

  since operations of distinct transactions on different data items commute.

- $S$ is a serializable schedule
Serializability: Commutativity

- Schedule
  
  $S$: $r_1(z) \ r_2(q) \ w_2(z) \ r_1(q) \ w_1(y)$

  is equivalent to the serial schedule $T1, T2$:
  
  $r_1(z) \ r_1(q) \ w_1(y) \ r_2(q) \ w_2(z)$

  since read operations of distinct transactions on the same data item commute.

- $S$ is not equivalent to $T2, T1$

  since read and write operations of distinct transactions on the same data item do not commute
Correctness of 2PL

- Intuition: Active transactions cannot have executed operations that do not commute, since locks required for non-commutative operations conflict.

- A schedule produced by a 2PL is serializable, since operations of concurrent transactions can always be reordered to produce a serial schedule.
2PL: Deadlock

T1
s1(Y)
r1(Y)
x1(X)
u1(Y)
r1(X)
X:=X+Y
w1(X)
u1(X)

T2
s2(X)
r2(X)
x2(Y)
u2(X)
r2(Y)
Y:=X+Y
w2(Y)
u2(Y)

T1 cannot proceed until T2 releases lock on X.
T2 cannot proceed until T1 releases lock on Y.
⇒ DEADLOCK
Conservative 2PL: Deadlock

In this case, the only possible schedules are serial schedules.

Locks must be acquired as a unit at beginning of transaction. Transaction cannot be holding locks while waiting on locks. Deadlock is not possible.
Conservative 2PL: Deadlock

**T1**
- $s_1(Y)$, $x_1(Z)$
- $r_1(Y)$
- $u_1(Y)$
- $r_1(Z)$
- $Z := Z + Y$
- $w_1(Z)$
- $u_1(Z)$

**T2**
- $s_2(X)$, $x_2(Y)$
- $r_2(X)$
- $u_2(X)$
- $r_2(Y)$
- $Y := X + Y$
- $w_2(Y)$
- $u_2(Y)$

T2 can proceed as soon as T1 releases lock on Y.

Concurrency is still possible under conservative 2PL.
Let deadlocks happen, then resolve the problem

**Wait-for graph**
- scheduler maintains a *wait-for graph*
  - arc from Tx to Ty indicates Tx is waiting for a lock held by Ty
  - when a transaction is blocked, it is added to the graph
  - a cycle in the wait-for-graph indicates deadlock
  - one transaction involved in the cycle is selected (victim) and rolled-back

**Timeout**
- abort any transaction that has been waiting for some set amount of time
- simple solution, but may be abort a transaction that could eventually proceed
Deadlock: Prevention

- **Locking policy**
  - Implement a CC policy that never allows deadlock to occur
  - Example: conservative 2PL

- **Waits-for cycle avoidance**
  - Use wait-for graph, but do not allow cycles to occur
  - Example: any transaction that would create a cycle is aborted
  - Other algorithms use timestamps to choose victim
Deadlock Victim Selection

- T1 tries to lock X, T2 holds lock on X
  - wait-die:
    if T1 is older than T2, T1 waits
    otherwise, T1 aborts
  - wound-wait:
    if T1 is older than T2, T2 aborts,
    otherwise, T1 waits
  - no-waiting:
    T1 aborts
  - cautious waiting:
    if T2 is waiting, T1 aborts,
    otherwise T1 waits
Starvation

- **Starvation**
  - A particular transaction consistently waits or gets restarted and never gets a chance to complete
  - Caused by deadlock victim selection policy
  - Inherent in all priority based scheduling mechanisms

- Example: Wound-Wait
  a younger transaction may always be aborted by a long running older transaction,
Multiversion concurrency control techniques

- Maintain versions of a data item and allocate correct version to a read operation of a transaction
- Read operation is never rejected.
- Requires significantly more storage to maintain versions
- Requires garbage collection to remove unneeded versions
Optimistic CC

- Serializability is tested at the time of commit

- **Read phase:**
  - transaction can read values of committed data items
  - writes are applied only to local copies (versions) of the data

- **Validation phase:**
  - Serializability is checked before transactions write their updates to the database

- **Write phase:**
  - if serializability check passed, write updates to database otherwise, abort (or restart)
Summary: 2PL

- prevents unwanted schedules by delaying conflicting operations
- guarantees equivalence to some serial schedule
- never requires transaction aborts due to conflict
- reduces concurrency
- may cause deadlock, livelock or starvation
Summary: Timestamp

- rejects transactions that request operations that are out of order
- order is determined by unique timestamps assigned to each transaction
- + guarantees equivalence to a particular serial schedule
- + cannot cause deadlock
- - may cause (cascading) aborts due to conflict
- - may cause starvation
Summary: Multiversion

- keep multiple versions of modified data items and selects the appropriate versions that each transaction sees
- + reads can proceed concurrently with conflicting writes
- + avoids cascading aborts due to conflict
- - requires additional storage space and maintenance
- - may cause deadlock
- - transaction commit may be delayed
Summary: Optimistic

- No checking is done while a transaction is executing
- All operations are performed on local copies of data items
- Validity of the transaction is checked at commit, invalid transactions are aborted
- + maximal concurrency
- + no possibility of deadlock
- - may cause aborts due to conflict (conflict can be tested using precedence graphs)
- - determination of validity is delayed until latest possible time
Summary: Multistate

- keep multiple values for everything for each transaction
- merge resulting states by resolving conflicts
- + useful for applications with long transactions (i.e. computer-aided design tools)
- + some applications never merge states (temporal databases)
- - conflict resolution may require user intervention
Database Recovery
Purpose of Database Recovery

- Bring the database into the *most recent consistent state* that existed prior to a failure.

- Preserve transaction properties
  - Atomicity, Consistency, Isolation and Durability

- Example:
  - bank database crashes before a fund transfer transaction completes
  - either one or both accounts may have incorrect values
  - database must be restored to the state before the transaction modified any of the accounts
Types of Failure

The database may become unavailable due to

- **Transaction failure:** Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
- **System failure:** System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- **Media failure:** Disk head crash, power disruption, etc.
Transaction Log

- Recovery from failures, may require
  - data values prior to modification: BFIM - BeFore Image
  - new value after modification: AFIM – AFter Image

- These values and other information are stored in a sequential file - a *transaction log*

- Sample log data:

<table>
<thead>
<tr>
<th>T ID</th>
<th>Back P</th>
<th>Next P</th>
<th>Operation</th>
<th>Data item</th>
<th>BFIM</th>
<th>AFIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>1</td>
<td>Begin</td>
<td>Data item</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>1</td>
<td>4</td>
<td>Write</td>
<td>X</td>
<td>X = 100</td>
<td>X = 200</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>8</td>
<td>Begin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>5</td>
<td>W</td>
<td>Y</td>
<td>Y = 50</td>
<td>Y = 100</td>
</tr>
<tr>
<td>T1</td>
<td>4</td>
<td>7</td>
<td>R</td>
<td>M</td>
<td>M = 200</td>
<td>M = 200</td>
</tr>
<tr>
<td>T3</td>
<td>0</td>
<td>9</td>
<td>R</td>
<td>N</td>
<td>N = 400</td>
<td>N = 400</td>
</tr>
<tr>
<td>T1</td>
<td>5</td>
<td>nil</td>
<td>End</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Update Options

- **Immediate Update:**
  As soon as a data item is modified in cache, the disk copy is updated

- **Deferred Update:**
  Modified data items in the cache are written to disk either after a transaction ends its execution, or after a fixed number of transactions have completed their execution
Data Caching

- Modified data items are first stored into a cache, and later flushed (written) to the disk

- The flushing is controlled by Dirty and Pin bits (flags)
  - Pin: A pinned data item cannot be flushed from the cache
  - Dirty (Modified): A data item has been modified and must eventually be flushed to disk
Cache Flushing

- **In-Place Update:**
  Modified values in cache replace actual values on disk

- **Shadow update:**
  Modified version of a data item does not overwrite disk copy but is written at a separate disk location
Undo and Redo

- To maintain atomicity, a transaction’s operations may need to be redone or undone.

- **Undo (roll-back):**
  - restore all BFIMs to disk (replace all AFIMs)

- **Redo (roll-forward):**
  - restore all AFIMs to disk
Roll-back Example

Three concurrent transactions and timeline before system crash
# Roll-back Example

Transaction log at time of crash

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>[start_transaction, T₃]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[read_item, T₃, C]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[write_item, T₃, B, 15, 12]</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[start_transaction, T₂]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[read_item, T₂, B]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[write_item, T₂, B, 12, 18]</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[start_transaction, T₁]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[read_item, T₁, A]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[read_item, T₁, D]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[write_item, T₁, D, 20, 25]</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[read_item, T₂, D]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[write_item, T₂, D, 25, 26]</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[read_item, T₃, A]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System crash
Roll-back Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>15</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

- [start_transaction, \(T_3\)]
- [read_item, \(T_3, C\)]
- [write_item, \(T_3, B, 15, 12\)]
- [start_transaction, \(T_2\)]
- [read_item, \(T_2, B\)]
- [write_item, \(T_2, B, 12, 18\)]
- [start_transaction, \(T_1\)]
- [read_item, \(T_1, A\)]
- [read_item, \(T_1, D\)]
- [write_item, \(T_1, D, 20, 25\)]
- [read_item, \(T_2, D\)]
- [commit_transaction, \(T_1\)]
- [write_item, \(T_2, D, 25, 26\)]
- [read_item, \(T_3, A\)]

Restored database state should be \(<30, 15, 40, 25\>

T3 is rolled-back, since it has not yet committed.

T2 is also rolled-back, since it read values written by T3.

T1 is has committed and is not dependent on other transaction, so it’s updates should remain in database.
Write-Ahead Logging

- The **Write-Ahead Logging (WAL)** protocol insures that log is consistent with database state at the time of a crash.

- WAL states that:
  - **For Undo:** Before a data item’s AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log.
  - **For Redo:** Before a transaction executes its commit operation, all its AFIMs must be written to the log.
  - In both cases, the log must be saved in stable storage, before the flush or commit is processed.
Recover Schemes

- **Steal = no pinning**
  - can flush data items to recover buffer space
  - smaller buffer space requirements
- **No-steal = pinning**
  - cannot flush pinned data items before xact commits
  - may require larger buffer space
- **Force**
  - dirty data items must be flushed when xact commits
- **No-force**
  - dirty data items do not have to be flushed at commit (but do need to be flushed eventually)
Recover Schemes

- The force/no-force and steal/no-steal protocols used determine the recovery scheme:

  Steal/No-Force $\rightarrow$ Undo/Redo
  Steal/Force $\rightarrow$ Undo/No-redo
  No-Steal/No-Force $\rightarrow$ No-undo/Redo
  No-Steal/Force $\rightarrow$ No-undo/No-redo
Checkpointing

- From time to time (randomly or under some criteria) database flushes its buffer to database disk to minimize the task of recovery.

- The following steps define a checkpoint operation:
  - Suspend execution of transactions temporarily.
  - Force write modified buffer data to disk.
  - Write a [checkpoint] record to the log, save the log to disk.
  - Resume normal transaction execution.

- During recovery redo or undo may be required for transactions appearing after [checkpoint] record.
Recovery: Deferred Update

- No-Undo/Redo
  - assume no-steal/force
  - during transaction, updates are only in cache and log
  - disk is updated at commit
- After reboot from a failure the log is used to redo all the transactions affected by this failure
  - No undo is required because no AFIM is flushed to the disk before a transaction commits
Recovery: Deferred Update

- T1 is already in checkpoint, no action required
- T2 and T3 are in log and must be redone
- T4 and T5 were not committed and can be ignored
Recovery: Immediate Update

- **Undo/No-redo**
  - assume steal/force
  - WAL of BFIMs required
  - AFIMs of a transaction are flushed to the disk at commit
- **undo** all active transactions during recovery
- no transaction needs to be **redone**
Recovery: Immediate Update

- **Undo/Redo**
  - assume steal/no-force
  - WAL of BFIMs and AFIMs required
  - checkpointing used
- **undo** all active transactions during recovery
- **redo** all transactions that committed since last checkpoint
The AFIM does not overwrite its BFIM but is recorded at another place on the disk.

At any time a data item has AFIM and BFIM (shadow copy of the data item) at two different places on the disk.

- NO-UNDO/NO-REDO
- Requires eventual merging of shadow to current database

X and Y: Shadow copies of data items
X' and Y': Current copies of data items