Transactions

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE855

◆ Transaction
» performs a single logical function
» all-or-none computation
❖ either all operations are executed or none
» must do so in the face of system failures

◆ Transaction execution
» start transaction
» series of read and write operations
» either a commit or abort operation
❖ commit: all transaction operations executed successfully
❖ abort: all transaction operations executed, and all changes undone (as if transaction never started)
Properties of Transactions

- atomic: actions occur indivisibly
- consistent: system invariants hold
  - for ex: conservation of money
  - note that inside transaction this is violated, but from outside, the transaction is indivisible
- isolated: transactions do not interfere with each other
  - aka serializable
  - looks as though all transactions done in some sequential order
- durable: once a transaction commits, results are permanent

Example of Serializable Transactions

```plaintext
Begin_transaction
x = 0;
End_transaction

Begin_transaction
x = 0;
x = x+1;
End_transaction

Begin_transaction
x = 0;
x = x+2;
End_transaction

Begin_transaction
x = 0;
x = x+3;
End_transaction
```
Transaction Primitives

- Transaction commands
  » begin-transaction
  » end-transaction
  » abort-transaction
    ✫ must return to state before the begin-transaction
    ✫ often referred to as “roll-back”
  » commit-transaction
    ✫ changes in transaction take effect to outside world

- Transaction operations
  » read
  » write
  » etc...

Transaction Example

- Suppose we have three transactions T1, T2, and T3
  » two data elements, A and B
  » scheduled in a round-robin scheduler
  » one operation per time slice

<table>
<thead>
<tr>
<th>T#</th>
<th>Ts</th>
<th>event1</th>
<th>event2</th>
<th>event3</th>
<th>event4</th>
<th>event5</th>
<th>event6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>20</td>
<td>A</td>
<td>r</td>
<td>A</td>
<td>w</td>
<td>A</td>
<td>r</td>
</tr>
<tr>
<td>T2</td>
<td>21</td>
<td>r</td>
<td>B</td>
<td>w</td>
<td>A</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>T3</td>
<td>22</td>
<td>A</td>
<td>B</td>
<td>r</td>
<td>A</td>
<td>r</td>
<td>w</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aw</td>
<td>Ar</td>
<td>Aw</td>
<td>Ar</td>
</tr>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td></td>
</tr>
</tbody>
</table>
Transaction Example (cont.)

◆ Objective: find some ordering in which atomicity is preserved
  » start out \( T_1 \rightarrow T_2 \rightarrow T_3 \)
    ◆ but \( T_1 \) reads \( A \) after \( T_3 \) writes
    ◆ now we have \( T_3 \rightarrow T_1 \)
    ◆ atomicity is not preserved
    ◆ abort \( T_1 \)
  » now try \( T_2 \rightarrow T_3 \rightarrow T_1 \)
    ◆ then \( T_2 \) writes \( A \) after \( T_3 \)'s write
    ◆ meaning \( T_3 \rightarrow T_2 \)
    ◆ abort \( T_2 \)
  » now try \( T_3 \rightarrow T_1 \rightarrow T_2 \)
    ◆ this works in the end...

Nested Transactions

◆ Transaction divided into sub-transactions
  » structured as a hierarchy
  » internal nodes are masters for its children
  » advantages:
    ◆ better performance: aborted sub-transactions do not abort masters
    ◆ increased concurrency: only need to lock sub-transactions
Nested Transactions (cont.)

- Aborting committed children
  » suppose a parent transaction starts several child transactions
  » one or more child commits
    ❖ only after committing is the child’s results visible to parent
    ❖ i.e. atomicity is preserved at child level
  » then parent aborts...
    ❖ but child already “committed”
  » parent abort must roll back all child transactions
    ❖ even if they have committed

Implementing Transactions

- Conceptually, a transaction is given a private workspace
  » consisting of all resources it has access to
  » before commit: all operations done to private workspace
  » after commit: changes are made to actual workspace (file system, etc.)
  » if the shadowed workspaces of more than one transaction intersects
    ❖ and one of them has a write operation
    ❖ then there is a conflict
    ❖ one of the transactions must be aborted
Implementing Transactions (cont.)

- Shadow blocks
  - problem: copying files to a private workspace is expensive!
  - so just copy the blocks that the transaction needs
  - copy index block for file instead of file
  - don’t need to copy blocks that are only read
  - demand-driven copying: only copy when a block is first modified
    - a kind of caching
  - write "shadowed" blocks on commit

Implementing Transactions
Writeahead Log

- Log consists of:
  - transaction name
  - data item name
  - old value
  - new value
  - Write log before performing write operations
  - onto non-volatile storage
  - Transaction log consists of:
    - <Ti start>
    - series of (Ti, x, old value, new value)
    - <Ti commits> or <Ti aborts>
  - Recovery procedures
    - undo(Ti): restores values written by Ti to old values
    - redo(Ti): sets all values written by Ti to new values
If Ti aborts:
   » execute undo(Ti)
If there is a system failure
   » can use redo(Ti) to make sure all updates are in place
      ♦ compare writeahead to actual value
      ♦ also use the log to proceed with the transaction
   » if an abort is necessary, use undo(Ti)
Note that the ‘commit’ operation must be done atomically
   » difficult when different machines, processes are involved

Coordinator is selected (transaction initiator)
   » Phase 1
      ♦ coordinator writes ‘prepare’ in log
      ♦ sends ‘prepare’ message to all processes involved in the commit (subordinates)
      ♦ subordinates write ‘ready’ (or ‘abort’) into log
      ♦ subordinates reply to coordinator
   » Phase 2
      ♦ coordinator logs received replies (or aborts)
      ♦ coordinator logs ‘commit’ and sends ‘commit’ message
      ♦ subordinates write ‘commit’ into their log
      ♦ do the commit
      ♦ send ‘finished’ message to coordinator
Implementing Transactions
Two-phase commit (cont.)

» If any subordinate cannot commit, abort transaction
  ◆ if, for example, the subordinate does not respond
» If all respond, ‘commit’ message makes transaction results stick
  ◆ i.e. now they are permanent
  ◆ can remove all transaction log entries, if desired
◆ Error recovery in two-phase commit uses log entries
  » determine when crash occurred
  » proceed from there
  » may need to repeat some messages

Concurrency Control

» Transactions may need to run simultaneously
  » transactions can conflict: one may write to a data item others want to read or write
  » need methods to synchronize concurrent access
◆ Concurrency control methods
  » locking
  » optimistic concurrency control
  » timestamps
**Locking**

- **Locks**
  - a semaphore of sorts
  - read locks: allow n read locks on a resource
  - write locks: no other lock is permitted

- **Two-Phase locking**
  - fine-grained locking can lead to deadlock
  - divide lock requests into two phases
    - growing phase: transaction obtains locks, may not release any
    - shrinking phase: once a lock is released, no locks can be obtained for rest of the transaction

**Disadvantage of two-phase locking**

- concurrency is reduced
- Deadlocks can occur in two-phase locking
  - resource ordering, etc. necessary to prevent deadlocks
Two-Phase Locking

- **Scenario 1**
  
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock R1</td>
<td>lock R1</td>
</tr>
<tr>
<td>...</td>
<td>lock R2</td>
</tr>
<tr>
<td>lock R2</td>
<td>...</td>
</tr>
<tr>
<td>unlock R1</td>
<td>unlock R1</td>
</tr>
<tr>
<td>unlock R2</td>
<td>unlock R2</td>
</tr>
</tbody>
</table>

- **Scenario 2**
  
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock R1</td>
<td>lock R2</td>
</tr>
<tr>
<td>...</td>
<td>lock R1</td>
</tr>
<tr>
<td>lock R2</td>
<td>...</td>
</tr>
<tr>
<td>unlock R1</td>
<td>unlock R1</td>
</tr>
<tr>
<td>unlock R2</td>
<td>unlock R2</td>
</tr>
</tbody>
</table>

Optimistic Concurrency Control

- **Conflicting transactions are rare**
  - therefore let a transaction make all changes
    - without checking for conflicts
  - at commit time, check for files that have changed since the transaction began
    - if so, abort
  - works best with shadowed implementations
    - initial changes made to private workspace
    - distributed transactions need some form of global time
      - for comparing time for file changes

- **Parallelism is maximized**
  - no waiting on locks
  - inefficient when an abort is needed
  - not a good strategy in systems with many potential conflicts
Timestamp Ordering

- Each transaction assigned a unique timestamp $TS(T_i)$
  - if $T_i$ enters system before $T_j$
  - $TS(T_i) < TS(T_j)$
- Each data item, $Q$, gets two timestamps:
  - $W$-timestamp($Q$): largest write timestamp
  - $R$-timestamp($Q$): largest read timestamp
- General concept
  - process transactions in a serial order
  - can use the same file, but must do it in order
  - therefore atomicity is preserved

For a read:

```c
if (TS(T_i) < W$\text{\text{-}}$timestamp($Q$))
{  reject read
   roll back and re-start $T_i$
}
else /* $TS(T_i) \geq W$-timestamp($Q$) */
{  execute read
   $R$-timestamp = max($R$-timestamp, $TS(T_i)$)
}
```

Timestamp ordering is deadlock-free

- essentially ordering the sequence of file accesses
- no cycles can result
Three transactions T1, T2, and T3

- Two data elements, A and B
- Scheduled in a round-robin scheduler
- One operation per time slice
- Use read and write timestamps

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
T1 w r i t e ( A) r e a d( A) w r i t e ( A)
T2 r e a d (A ) w r ite (B ) r e a d (B )
T3 w r i t e ( A)
```

**Timestamp Ordering Example**

```
T# Ts event1 event2 event3 event4 event5 event6 event7
T1 20 Aw Ar Bw Br
T2 21 Ar Bw Aw
T3 22 Aw
```

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```