Distributed Systems

Concurrency Control

Fall 2016
Problems with concurrent transactions

- Transactions are carried out concurrently for higher performance
  - Otherwise, painfully slow
- Serial Equivalence
  - Interleaved operations produce same effect as if transactions have been performed one at a time
  - Does not mean to actually perform one transaction at a time, as this would lead to horrible performance
- Two operations are in conflict if the final result depends on the order of execution
  - Value set by a write
  - Result of a read

Motivation

Read – Read  ≠  No conflict
Read – Write (or Write – Read)  ≠  Conflict!
Write – Write  ≠  Conflict!
Concurrency control

- Serialize access to objects
  - Each server is responsible for concurrency control on own objects
  - All servers are jointly responsible for concurrency control of conflicting transactions
- Ensure serially equivalent interleavings
- Maximize concurrency
  - Locks (wait for access)
  - Optimistic concurrency control (check for conflicts at the end)
  - Timestamp ordering (check to delay or reject)
Locks
Locks

- Need an object? Get a lock for it!
  - Read or write locks, or both (exclusive)
- Two-phase locking
  - Accumulate locks gradually, then release locks gradually
- Strict two-phase locking
  - Accumulate locks gradually, keep them all until completion
    Enables “strict” systems
- Granularity and tradeoffs
<table>
<thead>
<tr>
<th>Transaction</th>
<th>T :</th>
<th>Transaction</th>
<th>U :</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance()</td>
<td></td>
<td>balance = b.getBalance()</td>
<td></td>
</tr>
<tr>
<td>b.setBalance(bal * 1.1)</td>
<td></td>
<td>b.setBalance(bal * 1.1)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td></td>
<td>c.withdraw(bal/10)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>lock B</td>
<td>openTransaction</td>
<td>waits for T 's lock on B</td>
</tr>
<tr>
<td>bal = b.getBalance()</td>
<td></td>
<td>bal = b.getBalance()</td>
<td></td>
</tr>
<tr>
<td>b.setBalance(bal * 1.1)</td>
<td>lock A</td>
<td>b.setBalance(bal * 1.1)</td>
<td>lock C</td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>unlock A, B</td>
<td>c.withdraw(bal/10)</td>
<td>lock C</td>
</tr>
<tr>
<td>closeTransaction</td>
<td></td>
<td>closeTransaction</td>
<td>unlock B, C</td>
</tr>
</tbody>
</table>

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on Figure 16.14
Sharing locks

- Read locks can be shared
- Promote read lock to write lock is allowed if no other transactions require a lock
- Requesting a write lock when there are already read locks, or a read lock when there is already a write lock?
  - Wait until lock is available

<table>
<thead>
<tr>
<th>Lock already set</th>
<th>For one object</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>write</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>write</td>
</tr>
<tr>
<td>write</td>
<td>wait</td>
</tr>
</tbody>
</table>

Lock compatibility
Rules for strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.
Locks and nested transactions

- **Isolation**
  - From other sets of nested transactions
  - From other transactions in own set

- **Rules:**
  - Parents do not run concurrently with children
  - Children can temporarily acquire locks from ancestors
  - Parent inherits locks when child transactions commit
    - Locks are discarded if child aborts
  - Sub-transactions at each level are treated as flat transactions

There are also rules for acquiring and releasing locks
Big problem: Deadlocks

- Typical deadlock:
  - Transaction A waits for B, transaction B waits for A
- Deadlocks may arise in long chains
- Conceptually, construct a wait-for graph
  - Directed edge between nodes if one waits for the other
  - Cycles indicate deadlocks

Abort transaction(s) as needed
Handling deadlock

- **Deadlock prevention**
  - Acquire all locks from the beginning
  - Bad performance, not always possible

- **Deadlock detection**
  - As soon as a lock is requested, check if a deadlock will occur
  - Bad performance: avoid checking always
  - Must include algorithm for determining which transaction to abort

- **Lock timeouts**
  - Locks invulnerable for a certain time, then they are vulnerable
  - Leads to unnecessary aborts
    - Long-running transactions
    - Overloaded system
  - How to decide useful timeout value?
Distributed deadlock

- Local and distributed deadlocks
  - transaction locks are held in different servers,
  - introduce a coordinator to which each server forwards its wait-for graph
  - Coordinator able to produce a wait-for graph for the entire system
    - Information gathered at the coordinator regarding each server's wait-for graph is out of date?
  - Phantom deadlocks
  - Manager collects local wait-for information and constructs global wait-for graph
    - Single point of failure, bad performance, does not scale, what about availability, etc.
- Distributed solution – edge chasing or path pushing
  - Don't construct a global wait-for graph, instead only send probes
Phantom Deadlocks Example

(a) Release arrives before request

(b) Request arrives before release
Optimistic Concurrency control
Locks, drawbacks

- Overhead (even on read-only transactions)
  - Necessary only in the worst case
- Deadlock
  - Prevention reduces concurrency severely
  - Timeouts or detection
- Reduced concurrency in general
  - Locks need to be maintained until transactions end

Enter optimistic concurrency control
Optimistic Concurrency Control

- Assumes that conflicts are rare
  - Probability of multiple accesses to same object is low
  - Only need to worry about real conflicts
- Transaction phases:
  - **Working**
    - Transaction works with tentative data (read and write sets)
  - **Validation** (Upon completion)
    - Check if transaction may commit or abort
    - Conflict resolution
  - **Update**
    - Write tentative data from committed transactions to permanent storage
Validation

- Use conflict rules from earlier!
  - On overlapping transactions
- Validate one transaction at a time against others
- Transactions are numbered (not to be confused with IDs) as they enter the validation phase
- Only a single transaction at a time in update phase
- Backward or Forward validation

<table>
<thead>
<tr>
<th>Rule</th>
<th>( T_v )</th>
<th>( T_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>write read</td>
<td>( T_i ) must not read objects written by ( T_v )</td>
<td></td>
</tr>
<tr>
<td>read write</td>
<td>( T_v ) must not read objects written by ( T_i )</td>
<td></td>
</tr>
<tr>
<td>write write</td>
<td>( T_i ) must not read objects written by ( T_v ) and ( T_v ) must not read objects written by ( T_i )</td>
<td></td>
</tr>
</tbody>
</table>
Backward validation

- Check *read* set against *write* set of transactions that:
  - were active at the same time as the transaction currently being validated;
  - and
  - have already committed
- Transactions with only *write* set need not be checked
- If overlap is found, then current transaction must be aborted!

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design  Edn. 5 © Pearson Education 2012 – based on Figure 16.28
Backward validation of transaction $T_v$

```java
boolean valid = true;
for (int $i = startTn+1; i <= finishTn; $i++) {
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;
}
```
Forward validation

- Check **write** set against **read** set of transactions that are currently active
  - Note that read sets of active transactions may change during validation
- **read-only** transactions need not be checked
- If overlap is found, we can choose which transaction(s) to abort
  - Wait until conflicting transactions have finished
  - Abort conflicting active transactions
  - Abort transactions being validated
Forward validation of transaction $T_v$

```java
boolean valid = true;
for (int $T_{id} = active1; T_{id} <= activeN; T_{id}++){
    if (write set of $T_v$ intersects read set of $T_{id})$ valid = false;
}
```
Comparison of validation schemes

- **Size of read/write sets**
  - *Read* sets are usually bigger
  - Forward compares against “growing” *read* sets

- **Choice of transaction to abort**
  - Backward a single choice, Forward three choices
  - Linked to starvation

- **Overhead**
  - Backward requires storing old *write* sets
  - Forward may need to re-run each time the *read* set for any active transaction changes and must allow for checking new valid transactions
Optimistic Concurrency Control

- Superior to locking methods for systems where transaction conflict is highly unlikely
  
  - But starvation is a real issue

- Higher loads are an issue
Let us look at real systems!
2 PL in Spanner!

- Spanservers serve data to clients
- each spanserver implements a lock table to implement concurrency control.
  
  1 The lock table contains the state for two-phase locking
  
  - it maps ranges of keys to lock states
2 PL in Spanner!

- Reason
  - designed for long-lived transactions
  - perform poorly under optimistic concurrency control in the presence of conflicts
Optimistic concurrency control

- MediaWiki
  - Edits to wiki pages are rare compared to reads
    1. At least for Wikipedia!
  - Makes little sense to implement locking
  - Starvation is no issue!
Timestamp ordering
Overview

- Avoids locks, relies on timestamps
- Transactions are assigned timestamps when they start
  - Timestamps are assigned to all read and write accesses that a transaction makes
- Read and write access is granted according to timestamp order
  - validated when carried out
  - Requests are totally ordered
  - Serial execution of transactions
  - Transactions are aborted if validation is unsuccessful
Ordering rule

- Based on operation conflicts
  - Writes are valid only if the object was last read or written by earlier transactions
  - Reads are valid only if the object was last written by an earlier transactions
- Transactions can access an object concurrently
  - Writes on tentative versions until committed
  - Writes may be performed after `closeTransaction()`
  - Reads must wait for earlier transactions to finish (no deadlock)
Details

- Tentative versions are created when writes are accepted
  - Write timestamp set to transaction timestamp
- Reads are directed to a version according to timestamp
  - The earliest version
  - Transaction timestamp is added to read timestamps
- For commits:
  - Tentative version becomes the object (values)
  - Tentative version timestamps become the objects’ timestamps

---

**Operation conflicts for timestamp ordering**

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. write</td>
<td>read</td>
<td>$T_c$ must not write an object that has been read by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.</td>
</tr>
<tr>
<td>2. write</td>
<td>write</td>
<td>$T_c$ must not write an object that has been written by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
<tr>
<td>3. read</td>
<td>write</td>
<td>$T_c$ must not read an object that has been written by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
</tbody>
</table>
A write is accepted if $(\text{transaction } T_c, \text{ object } D)$:

- if $(T_c \geq \text{ maximum read timestamp on } D \land T_c > \text{ write timestamp on committed version of } D)$
  - perform write operation on tentative version of $D$ with write timestamp $T_c$
- else /* too late */
  - Abort transaction $T_c$

When a write is accepted a new tentative version is created with timestamp $T_c$

Writes that arrive too late are aborted

- A transaction with a later timestamp has already operated on the object
Example: write operations

(a) \( T_3 \) write

Before

\[ T_2 \]

After

\[ T_2 \quad T_3 \]

Time

(b) \( T_3 \) write

Before

\[ T_1 \quad T_2 \]

After

\[ T_1 \quad T_2 \quad T_3 \]

Time

Key:

\[ T_i \]

Committed

\[ T_i \]

Tentative

Object produced by transaction \( T_i \) (with write timestamp \( T_i \))

\[ T_1 < T_2 < T_3 < T_4 \]

(c) \( T_3 \) write

Before

\[ T_1 \quad T_4 \]

After

\[ T_1 \quad T_3 \quad T_4 \]

Time

(d) \( T_3 \) write

Before

\[ T_4 \]

Transaction aborts

After

\[ T_4 \]

Time

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 - based on Figure 16.30
A read is accepted if \((\text{transaction } T_c, \text{ object } D)\):

\[
\text{if } (T_c > \text{write timestamp on committed version of } D) \{
    \text{let } D_{\text{selected}} \text{ be the version of } D \text{ with the maximum write timestamp } \leq T_c \\
    \text{if } (D_{\text{selected}} \text{ is committed}) \\
    \quad \text{perform read operation on the version } D_{\text{selected}} \\
    \text{else} \\
    \quad \text{Wait until the transaction that made version } D_{\text{selected}} \text{ commits or aborts} \\
    \quad \text{then reapply the read rule} \\
\} \text{ else} \\
\text{Abort transaction } T_c
\]

Reads that arrive too early need to wait for the earlier transaction to complete (aborts dirty reads)

Reads that arrive too late are aborted
Example: read operations

(a) T₃ read

(b) T₃ read

(c) T₃ read

(d) T₃ read

Key:

- Tᵢ: Committed
- Tᵢ: Tentative

Transaction aborts

object produced by transaction Tᵢ (with write timestamp Tᵢ)
Tᵢ < T₂ < T₃ < T₄
Combined example

Timestamp ordering

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RTS</td>
<td>WTS</td>
<td>RTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{}</td>
<td>S</td>
<td>{}</td>
</tr>
</tbody>
</table>

openTransaction

bal = b.getBalance()

b.setBalance(bal*1.1)

openTransaction

bal = b.getBalance()

wait for T

a.withdraw(bal/10)

commit

bal = b.getBalance()

b.setBalance(bal*1.1)

c.withdraw(bal/10)

S<T<U

---

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on Figure 16.32
Required reading!

- Chandy, Misra and Haas paper on distributed deadlock detection!
- Office hours:
  - Today until 12:00
  - Tomorrow after Lunch
  - Friday before Lunch
Drop me a line an hour before you come on any other day!