CompSci 516
Data Intensive Computing Systems

Lecture 15
Transactions
– Concurrency Control

Instructor: Sudeepa Roy
Announcements

• No class on 3/15 and 3/17 (spring break)
  – there is a class this Thursday 3/10 before the break

• HW3 due after spring break
  – 03/23 (Wed), 11:55 pm
  – Note that the main goal is NOT to solve the problems or to learn a new language..
  – …but to learn a new framework (MapReduce with inbuilt Map and Reduce functions) with some simple tasks
  – it can be useful when you are dealing with “BIG DATA” (e.g. Twitter data), and running complex (even simple) tasks on multiple machines
Reading Material

• [RG]
  – Chapter 17.5.1, 17.5.3, 17.6

• [GUW]
  – Chapter 18.8, 18.9

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
What we learnt in Lectures 12 and 13

• Transaction
  – R₁(A), W₂(A), ....
  – Commit C₁, abort A₁
  – Lock/unlock: S₁(A), X₁(A), US₁(A), UX₁(A)

• ACID properties
  – what they mean, whose responsibility to maintain each of them

• Conflicts: RW, WR, WW

• 2PL/Strict 2PL
  – all lock acquires have to precede all lock releases
  – Strict 2PL: release X locks only after commit or abort
What we learnt in Lectures 12 and 13

- Serial schedule
- Serializable schedule (why do we need them?)
- Conflicting actions
- Conflict-equivalent schedules
- Conflict-serializable schedule
- View-serializable schedule (relaxation)
- Conflict Serializability $\Rightarrow$ View Serializability $\Rightarrow$ Serializability
- Recoverable schedules
What we learnt in Lectures 12 and 13

• Dependency (or Precedence) graphs
  – their relation to conflict serializability (by acyclicity)
  – their relation to Strict 2PL
• Lock management basics
• Deadlocks
  – detection
    • waits-for graph has cycle, or timeout
    • what to do if deadlock is detected
  – prevention
    • wait-die and wound-wait
Today’s topics

• Dynamic databases and Phantom problem (17.5.1)
• Multiple—granularity locking (17.5.3)
• Optimistic concurrency control (17.6.1)
• Timestamp-based concurrency control (17.6.2)
• Multi-version concurrency control (17.6.3)
Dynamic Database and Phantom Problem
Dynamic Databases

• If we relax the assumption that the DB is a fixed collection of objects
• even Strict 2PL will not assure serializability
Dynamic Databases

• If we relax the assumption that the DB is a fixed collection of objects

• Then even Strict 2PL will not assure serializability

• "Phantom Problem" in dynamic databases
Example: Phantom Problem

• T1 wants to find oldest sailors in rating levels 1 and 2
  – Suppose the oldest at rating 1 has age 71
  – Suppose the oldest at rating 2 has age 80
  – Suppose the second oldest at rating 2 has age 63

• Another transaction T2 intervenes:
  – Step 1: T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (age = 71)
  – Step 2: Next, T2 inserts a new sailor onto a new page (rating = 1, age = 96)
  – Step 3: T2 locks pages with rating = 2, deletes oldest sailor with rating = 2 (age = 80), commits, releases all locks
  – Step 4: T1 now locks all pages with rating = 2, and finds oldest sailor (age = 63)

• No consistent DB state where T1 is “correct”
  – T1 found oldest sailor with rating = 1 before modification by T2
  – T1 found oldest sailor with rating = 2 after modification by T2
What was the problem?

• Conflict serializability guarantees serializability only if the set of objects is fixed

• Problem:
  – T1 implicitly assumed that it has locked the set of all sailor records with rating = 1
  – Assumption only holds if no sailor records are added while T1 is executing
  – Need some mechanism to enforce this assumption

• Index locking and predicate locking
Index Locking

• If there is a dense index on the rating field using Alt. (2), T1 should lock the index page containing the data entries with rating = 1
  – If there are no records with rating = 1, T1 must lock the index page where such a data entry would be, if it existed

• If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added
  – to ensure that no new records with rating = 1 are added
Predicate Locking

• Grant lock on all records that satisfy some logical predicate, e.g. rating = 1 or, age > 2*salary

• Index locking is a special case and an efficient implementation of predicate locking
  – e.g. Lock on the index pages for records satisfying rating = 1

• The general predicate locking has a lot of locking overhead and so not commonly used
Multiple-granularity Locking
DB Objects may contain other objects

• A DB contains several files
• A file is a collection of pages
• A page is a collection of records/tuples
Carefully choose lock granularity

• If a transaction needs most of the pages
  – set a lock on the entire file
  – reduces locking overhead

• If only a few pages are needed
  – lock only those pages

• Need to efficiently ensure no conflicts
  – e.g. a page should not be locked by T1 if T2 already holds the lock on the file
New Lock Modes & Protocol

- Allow transactions to lock at each level, but with a special protocol using new “intention locks”:
  - Before locking an item (S or X), transaction must set “intention locks” (IS or IX) on all its ancestors
  - For unlock, go from specific to general (i.e., bottom-up)
    - otherwise conflicting lock possible at root

### Conflicting Locks

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IS</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

other tr. cannot have IX or X

other tr. cannot have any other lock
SIX mode = S + IX

- Common situation: a transaction needs to read an entire file and modify a few records
  - S lock
  - IX lock (to subsequently lock)
    some containing objects in X mode

- Obtain a SIX lock
  - conflict with either S or IX

Other tr. cannot have IX or X

Other tr. cannot have any other lock

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Approaches to CC other than locking
Approaches to Concurrency Control (CC)

• Lock-based CC
  – (so far)
• Optimistic CC
  – today
• Time-stamp-based CC
  – today
• Multi-version CC
  – today

uses “timestamps” in some way
Timestamp

• Each transaction gets a unique timestamp

• e.g.
  – system’s clock value when it is issued by the scheduler (assume one transactions issued on one tick of the clock)
  – or a unique number given by a counter (incremented after each transaction)
Locking is a pessimistic approach to CC

• Locking is a conservative approach in which conflicts are prevented

• Either uses “blocking” (delay) or abort
  – note the several usages of a “block”!

• Disadvantages of locking:
  – Lock management overhead
  – Deadlock detection/resolution
  – Lock contention for heavily used objects

• If only light contention for data objects, still the overhead of following a locking protocol is paid
Optimistic CC
A second approach to CC: Optimistic CC (Kung-Robinson)

• If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before transactions commit

• Premise:
  – most transactions do not conflict with other transactions
  – be as permissive as possible in allowing transactions to execute
Kung-Robinson Model

• Transactions have three phases:
  
 1. **READ**: Read from the database, but make changes to "private copies" of objects (assume private workspace)

 2. **VALIDATE**: When decide to commit, also check for conflicts with concurrently executing transactions
   - if a possible conflict, abort, clear private workspace, restart

 3. **WRITE**: If no conflict, make local copies of changes public (copy them into the database)
Validation

- Test conditions that are sufficient to ensure that no conflict occurred

- Each transaction Ti is assigned a numeric id
  - Use a timestamp TS(Ti)

- Transaction ids assigned at end of READ phase, just before validation begins

- Validation checks whether the timestamp ordering has an equivalent serial order
Notation

- **TS(Ti):** Transaction id or timestamp of Ti BEFORE the validation step starts

- **ReadSet(Ti):** Set of objects read by transaction Ti

- **WriteSet(Ti):** Set of objects modified by transaction Ti
Validation Tests

• To validate Tj
  – for each committed transactions Ti
  – such that \( TS(Ti) < TS(Tj) \)
  – one of the three validation tests (TEST 1, TEST 2, TEST 3) must be satisfied
  – (see the tests next)

• Ensures that Tj-s modifications are not visible to Ti

• Check yourself: No RW, WR, WW conflicts if any of these tests satisfy
Test 1

- For all i and j such that $TS(Ti) < TS(Tj)$, check that Ti completes (all three phases) before Tj begins.

- Tj sees some changes by Ti.
- But they execute completely in serial order.
Test 2

- For all i and j such that $TS(T_i) < TS(T_j)$, check that:
  - $T_i$ completes before $T_j$ begins its Write phase +
  - $WriteSet(T_i) \cap ReadSet(T_j)$ is empty

- Allows $T_j$ to read objects while $T_i$ is still modifying objects
- But no conflict because $T_j$ does not read any object modified by $T_i$
- $T_j$ can overwrite some writes by $T_i$
Test 3

• For all $i$ and $j$ such that $T_i < T_j$, check that:
  
  – $T_i$ completes Read phase before $T_j$ completes its Read +
  
  – $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)$ is empty +
  
  – $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j)$ is empty

\[ \begin{align*}
\text{Ti} & \quad \text{R} & \text{V} & \text{W} \\
\text{Tj} & \quad \text{R} & \text{V} & \text{W}
\end{align*} \]

• Allows $T_i$ and $T_j$ write objects at the same time
• More overlap than Test 2
• But the sets of objects written cannot overlap
Comments on Serial Validation

• List of objects written/read by each transaction has to be maintained

• While one transaction is validating, no transaction can commit
  – otherwise some conflicts may be missed

• Assignment of transaction id, validation, and the Write phase are inside a critical section
  – i.e., Nothing else goes on concurrently
  – If Write phase is long, major drawback

• The write phase of a validated transactions must be completed before other tr. s are validated
  – i.e. changes should be reflected to the DB from private workspace

• Optimization for Read-only transactions:
  – Don’t need critical section (because there is no Write phase)
Overheads in Optimistic CC

• Must record read/write activity in ReadSet and WriteSet per transaction
  – Must create and destroy these sets as needed

• Must check for conflicts during validation, and must make validated writes `global’’
  – Critical section can reduce concurrency

• Optimistic CC restarts transactions that fail validation
  – Work done so far is wasted; requires clean-up
Optimistic CC vs locking

• If there are few conflicts and validation is efficient
  – optimistic CC is better than locking

• If many conflicts
  – cost of repeatedly restarting transactions hurts performance significantly
Timestamp-based CC
A third approach to CC

So far...

• **Lock-based CC**
  – conflicting actions of different transactions are ordered by the order in which locks are obtained
  – locking protocols ensure serializability

• **Optimistic CC**
  – A timestamp ordering is imposed on transactions
  – Validation checks that all conflicting transactions occurred in the same order

• **Next: Timestamp-based CC**
  – another use of timestamp
Main Idea:

- **Give each object O**
  - a read-timestamp RT(O), and
  - a write-timestamp WT(O)
    - RG uses RTS/WTS, GUW uses RT/WT, any of these is fine

- **Give each transaction T**
  - a timestamp TS(T) when it begins:

  - **If**
    - action ai of Ti conflicts with action aj of Tj,
    - and TS(Ti) < TS(Tj)
  - **then**
    - ai must occur before aj
  - **Otherwise, abort and restart violating transaction**
When T wants to read Object O

- If TS(T) < WT(O)
  - this violates timestamp order of T w.r.t. writer of O
  - So, abort T and restart it with a new, larger TS
  - Note: If restarted with same TS, T will fail again
    - Revisit: Contrast use of timestamps in 2PL for deadlock prevention (same timestamps were used)

- If TS(T) > WT(O):
  - Allow T to read O
  - Reset RT(O) to \( \max(\text{RT}(O), \text{TS}(T)) \)

- Change to RT(O) on reads must be written to disk
  - This and restarts represent overheads
When T wants to Write Object O

• If $\text{TS}(T) < \text{RT}(O)$
  – this violates timestamp order of $T$ w.r.t. writer of $O$
  – abort and restart $T$

• If $\text{TS}(T) < \text{WT}(O)$
  – violates timestamp order of $T$ w.r.t. writer of $O$
  – Naïve approach: abort and restart $T$
    • (Better approach: use Thomas Write Rule (next))

• Else
  – allow $T$ to write $O$
Thomas Write Rule

- If $\text{TS}(T) < \text{WT}(O)$
  - violates timestamp order of $T$ w.r.t. writer of $O$

**Thomas Write Rule:**
- But we can safely ignore such outdated writes
- no need to restart $T$
- $T$’s write is effectively followed by another write, with no intervening reads
- Allows some serializable, but **NOT** conflict serializable schedules

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>W(A) Commit</td>
</tr>
<tr>
<td>W(A) Commit</td>
<td></td>
</tr>
</tbody>
</table>

A Serializable schedule that is not conflict-serializable
Unfortunately, unrecoverable schedules are allowed:

- \( \text{TS}(T_1) = 1 \)
- \( \text{TS}(T_2) = 2 \)

- **Timestamp CC can be modified to allow only recoverable schedules:**
  - **Buffer all writes** until writer commits (but update \( \text{WT}(O) \) when the write is allowed.)
  - **“Block” readers** \( T \) (where \( \text{TS}(T) > \text{WT}(O) \)) until writer of \( O \) commits
  - a full example from GUW in the next lecture

- Similar to writers holding X locks until commit, but still not quite 2PL
Multiversion CC
A fourth approach to CC

• Multiversion CC
  – another way of using timestamps
  – ensures that a transaction never has to wait to read an object

• The idea is to make several copies of each DB object
  – each copy of each object has a write timestamp

• Ti reads the most recent version whose timestamp precedes TS(Ti)
Multiversion Timestamp CC

• **Idea:** Let writers make a “new” copy while readers use an appropriate “old” copy:

MAIN SEGMENT (Current versions of DB objects)  

VERISON POOL (Older versions that may be useful for some active readers.)

Readers are always allowed to proceed
  – But may be “blocked” until writer commits.
Multiversion CC (Contd.)

• Each version of an object has
  – its writer’s TS as its WT, and
  – the timestamp of the transaction that most recently read this version as its RT

• Versions are chained backward
  – we can discard versions that are “too old to be of interest”

• Each transaction is classified as Reader or Writer.
  – Writer *may* write some object; Reader never will
  – Transaction declares whether it is a Reader when it begins
Reader Transaction

• For each object to be read:
  – Finds newest version with $WT < TS(T)$
  – Starts with current version in the main segment and chains backward through earlier versions

• Assuming that some version of every object exists from the beginning of time, Reader transactions are never restarted
  – However, might block until writer of the appropriate version commits
Writer Transaction

- To read an object, follows reader protocol
- To write an object:
  - must make sure that the object has not been read by a "later" transaction
  - Finds newest version $V$ s.t. $WT < TS(T)$.
- If $RT(V) < TS(T)$
  - T makes a copy $CV$ of $V$, with a pointer to $V$, with $WT(CV) = TS(T)$, $RT(CV) = TS(T)$
  - Write is buffered until T commits; other transactions can see TS values but can’t read version $CV$
- Else
  - reject write

Duke CS, Spring 2016

CompSci 516: Data Intensive Computing Systems
Summary

• Understand the reason for “Phantom Problem” and why serializability/2PL fails

• Understands new requirements and mechanisms for multiple-granularity locks

• Note the key ideas for three timestamp-based alternative approaches (to Lock-based approaches) to CC
  – Optimistic: validation tests
  – Timestamp: RT(O) & WT(O) on each object O
  – Multiversion: multiple versions of each object O with different WT and RT

• Note: a new action (block or delay) in addition to commit or abort

• Next lecture: full examples on some of these approaches from GUW (lecture slides will be sufficient for the exams and hws)