CSE 544
Principles of Database Management Systems

Alvin Cheung
Fall 2015
Lecture 12 – Transactions: Concurrency Control (Part 2 aka the Interesting Stuff)
Announcements

• Project milestone report due next Wednesday
  – See project page for details

• HW3 due next Thursday

• No lecture and OH next Tuesday

• Today: finish discussion on concurrency control
References

• **Concurrency control and recovery.**

• **Database management systems.**
  Ramakrishnan and Gehrke.
  Third Ed. *Chapters 16 and 17.*
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Motivating Example

UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3

SELECT sum(money)
FROM Budget

Would like to treat each group of instructions as a unit
Definition

• **A transaction** = one or more operations, (seemingly) single real-world transition

• **Examples**
  – Transfer money between accounts
  – Purchase a group of products
  – Register for a class (either waitlist or allocated)
  – What else?
ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs
- **Consistency**: A transaction as a whole does not violate integrity constraints
- **Isolation**: Transactions appear to execute one after the other in sequence
- **Durability**: If a transaction commits, its changes will survive failures
Types of Problems: Summary

• **Concurrent execution problems**
  - **Write-read conflict: dirty read (includes inconsistent read)**
    • A transaction reads a value written by another transaction that has not yet committed
  - **Read-write conflict: unrepeatable read**
    • A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
  - **Write-write conflict: lost update**
    • Two transactions update the value of the same object. The second one to write the value overwrite the first change

• **Failure problems**
  - DBMS can crash in the middle of a series of updates
  - Can leave the database in an inconsistent state
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
A Serial Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td></td>
<td>t := t + 100</td>
<td>t := t + 100</td>
</tr>
<tr>
<td></td>
<td>WRITE(A, t)</td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td></td>
<td>READ(A, s)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td></td>
<td>s := s * 2</td>
<td>s := s * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s)</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Serializable Schedule

- A schedule is *serializable* if it is equivalent to a serial schedule
A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

Notice:
This is NOT a serial schedule
Notation

$T_1: r_1(A); w_1(A); r_1(B); w_1(B)$
$T_2: r_2(A); w_2(A); r_2(B); w_2(B)$
Serializable Execution

- **Serializability**: interleaved execution has same effect as some serial execution

- **Schedule** of two transactions (Figure 1)
  
  \[
  r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow \\
  \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0
  \]

- **Serializable schedule**: equiv. to serial schedule
  
  \[
  r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_0[B] \rightarrow \\
  \rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[B] \rightarrow c_1
  \]
Conflict Serializability

Conflicts: (aka bad things happen if swapped)

Two actions by same transaction $T_i$:

Two writes by $T_i$, $T_j$ to same element

Read/write by $T_i$, $T_j$ to same element
Conflict Serializability

- A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions.

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
The Precedence Graph Test

Is a schedule conflict-serializable?
Simple test:
• Build a graph of all transactions $T_i$

• Edge from $T_i$ to $T_j$ if $T_i$ makes an action that conflicts with one of $T_j$ and comes first

• Fact: if the graph has no cycles, then it is conflict serializable!
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]

This schedule is NOT conflict-serializable
Conflict Serializability

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X); \]

Lost write

\[ w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X); \]

Equivalent, but can’t swap
Scheduler

- The scheduler is the module that schedules the transaction’s actions, ensuring serializability

- How? We discuss three techniques in class:
  - Locks
  - Timestamps
  - Validation
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Locking Scheduler

Simple idea:
• Each element has a unique lock
• Each transaction must first acquire the lock before reading/writing that element
• If lock is taken by another transaction, then wait
• The transaction must release the lock(s)
Notation

\[ \text{\texttt{I}}_i(A) = \text{transaction } T_i\text{ acquires lock for element } A \]

\[ \text{\texttt{u}}_i(A) = \text{transaction } T_i\text{ releases lock for element } A \]
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); \text{READ}(A, t) )</td>
<td>( L_2(A); \text{READ}(A,s) )</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s \times 2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A, t); U_1(A); L_1(B) )</td>
<td>( \text{WRITE}(A,s); U_2(A); L_2(B); \text{DENIED}... )</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>... \text{GRANTED}; \text{READ}(B,s)</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s \times 2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B,t); U_1(B); )</td>
<td>( \text{WRITE}(B,s); U_2(B); )</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule.
Is this enough?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A, t)</td>
<td>L₂(A); READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t); U₁(A);</td>
<td>WRITE(A, s); U₂(A);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s); U₂(B);</td>
</tr>
<tr>
<td>L₁(B); READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t); U₁(B);</td>
<td></td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability !!!
Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must precede all unlock requests
- This ensures conflict serializability! (why?)
Example: 2PL transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B); \text{READ}(A, t)$</td>
<td>$L_2(A); \text{READ}(A, s)$</td>
</tr>
<tr>
<td>$t := t + 100$</td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td>WRITE($A$, $t$); $U_1(A)$</td>
<td>WRITE($A$, $s$);</td>
</tr>
<tr>
<td>READ($B$, $t$)</td>
<td>$L_2(B); \text{DENIED}$...</td>
</tr>
<tr>
<td>$t := t + 100$</td>
<td></td>
</tr>
<tr>
<td>WRITE($B$, $t$); $U_1(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Now it is conflict-serializable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...$\text{GRANTED}$; \text{READ}(B, s)$</td>
</tr>
<tr>
<td></td>
<td>$s := s \times 2$</td>
</tr>
<tr>
<td></td>
<td>WRITE($B$, $s$); $U_2(A); U_2(B)$;</td>
</tr>
</tbody>
</table>
Example with Multiple Transactions

Equivalent to each transaction executing entirely the moment it enters shrinking phase
What about Aborts?

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?
Example with Abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); L₁(B); READ(A, t)</td>
<td>L₂(A); READ(A, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(A, t); U₁(A)</td>
<td>WRITE(A, s);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); DENIED…</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t); U₁(B);</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>…GRANTED; READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s); U₂(A); U₂(B);</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed
  - Also called “long-duration locks”

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit

- Avoids cascading rollbacks
Deadlock

- Transaction $T_1$ waits for a lock held by $T_2$;
- But $T_2$ waits for a lock held by $T_3$;
- While $T_3$ waits for . . . .
- . . .
- . . .and $T_{73}$ waits for a lock held by $T_1$ !!

- A deadlock is when two or more transactions are waiting for each other to complete
Handling Deadlock

• **Deadlock avoidance**
  – Acquire locks in pre-defined order
  – Acquire all locks at once before starting

• **Deadlock detection**
  – Timeouts (but hard to pick the right threshold)
  – Wait-for graph
    • What commercial systems use (they check graph periodically)
Lock Modes

• **S** = shared lock (for READ)
• **X** = exclusive lock (for WRITE)

• **U** = update lock
  – Initially like **S**
  – Later may be upgraded to **X**

• **I** = increment lock (for A := A + something)
  – Increment operations commute
Lock Granularity

• **Fine granularity locking** (e.g., tuples)
  – High concurrency
  – High overhead in managing locks

• **Coarse grain locking** (e.g., tables)
  – Many false conflicts
  – Less overhead in managing locks

• **Alternative techniques**
  – Hierarchical locking (and intentional locks) [commercial DBMSs]
  – Lock escalation
Phantom Problem

• A “phantom” is a tuple that is invisible during part of a transaction execution but not all of it.

• Example:
  – T0: reads list of books in catalog
  – T1: inserts a new book into the catalog
  – T2: reads list of books in catalog
    • New book will appear!

• How can this occur?
• Depends on locking details (eg, granularity of locks)
• Can’t lock a tuple that doesn’t exist yet
Dealing with Phantoms: Predicate Locks

• Lock predicates rather than actual database elements
  – “lock all books that have createTime > T”
  – Two predicates $p$ and $p'$ are compatible iff no tuple can satisfy both at the same time

• Issue: very expensive to implement
  – NP-hard to determine if predicates are compatible with each other
  – What if DB has hidden predicates (e.g., functional dependencies) that make $p$ and $p'$ incompatible?
Dealing with Phantoms: Granular Locks

• Implement multi-level locking
  – Tuple
  – Table
  – Entire database

• Allow transactions to lock at any granularity
  – Lock tuples if reading
  – Lock the entire table if inserting new records
  – Need a hierarchy of locks: table lock > tuple lock

• Issue: can cause many deadlocks among transactions
Dealing with Phantoms: Intent Locks

• Reduce possibility of deadlocks with three lock modes:
  – Shared
  – Exclusive
  – Intent

• Intent: transaction will be locking at finer granularity

• Lock compatibilities:

<table>
<thead>
<tr>
<th>Request \ Current mode</th>
<th>None</th>
<th>Intent</th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Intent</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Exclusive</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Degrees of Isolation

- Isolation level “serializable” (i.e. ACID)
  - Gold standard
  - Requires strict 2PL and predicate locking
  - But often too inefficient
  - Imagine there are only a few update operations and many long read operations

- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often default)
  - Sometimes are hard to understand
Degrees of Isolation

• **Four levels of isolation**
  – All levels use **long-duration exclusive locks**
  – **READ UNCOMMITTED**: no read locks
  – **READ COMMITTED**: short duration read locks
  – **REPEATABLE READ**:
    • Long duration read locks on individual items
  – **SERIALIZABLE**:
    • All locks long duration and lock predicates

• **Trade-off: consistency vs concurrency**
• Commercial systems give **choice** of level + **others**
The Tree Protocol

• An alternative to 2PL, for tree structures
• E.g. B+ trees (the indexes of choice in databases)

• Because
  – Indexes are hot spots!
  – 2PL would lead to great lock contention
  – Also, unlike data, the index is not directly visible to transactions
  – So only need to guarantee that index returns correct values
The Tree Protocol

Rules:

• A lock on a node A may only be acquired if the transaction holds a lock on its parent B
• Nodes can be unlocked in any order (no 2PL necessary)
• Cannot relock a node for which already released a lock
• “Crabbing”
  – First lock parent then lock child
  – Keep parent locked only if may need to update it
  – Release lock on parent if child is not full
• The tree protocol is NOT 2PL, yet ensures conflict-serializability!
• (More in the R&G)
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Locking vs Optimistic

• Locking prevents unserializable behavior from occurring: it causes transactions to wait for locks

• Optimistic methods assume no unserializable behavior will occur: they abort transactions if it does

• Locking typically better in case of high levels of contention; optimistic better otherwise
Optimistic Concurrency Control

**Timestamp-based technique**
- Each object, O, has read and write timestamps: RTS(O) and WTS(O)
- Each transaction, T, has a timestamp TS(T)
- **INVARIANT:** Timestamp order defines serialization order

**Transaction wants to read object O**
- If TS(T) < WTS(O) abort
- Else read and update RTS(O) to larger of TS(T) or RTS(O)

**Transaction wants to write object O**
- If TS(T) < RTS(O) abort
- If TS(T) < WTS(O) ignore my write and continue (Thomas Write Rule)
- Otherwise, write O and update WTS(O) to TS(T)
Optimistic Concurrency Control

Timestamp-based technique

• What about aborts? Need to add a commit bit C to each element
• Read dirty data:
  – T wants to read X, and WT(X) < TS(T)
  – If C(X) = false, T needs to wait for it to become true in case previous writer aborts
• Write dirty data:
  – T wants to write X, and WT(X) > TS(T)
  – If C(X) = false, T needs to wait for it to become true in case of abort

• Bottom line: When T requests r(X) or w(X), scheduler examines RT(X), WT(X), C(X), and decides one of:
  – To grant the request, or
  – To rollback T (and restart with later timestamp)
  – To delay T until C(X) = true
Optimistic Concurrency Control

Multiversion-based technique

• Object timestamps: RTS(O) & WTS(O); transaction timestamps TS(T)

• Transaction can read most recent version that precedes TS(T)
  – When reading object, update RTS(O) to larger of TS(T) or RTS(O)

• Transaction wants to write object O
  – If TS(T) < RTS(O) abort
  – Otherwise, create a new version of O with WTS(O) = TS(T)

• Common variant (used in commercial systems)
  – To write object O only check for conflicting writes not reads
  – Use locks for writes to avoid aborting in case conflicting transaction aborts
Optimistic Concurrency Control

Validation-based technique

• **Phase 1: Read**
  – Transaction reads from database and writes to a private workspace
  – Each transaction keeps track of its read set RS(T) and write set WS(T)

• **Phase 2: Validate**
  – At commit time, system performs validation using read/write sets
  – Validation checks if transaction could have conflicted with others
    • Each transaction gets a timestamp
    • Check if timestamp order is equivalent to a serial order
  – If there is a potential conflict: abort

• **Phase 3: Write**
  – If no conflict, transaction changes are copied into database
Snapshot Isolation

- A type of multiversion concurrency control algorithm
- Provides yet another level of isolation

- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server 2005

- Prevents many classical anomalies BUT…
- Not serializable (!), yet ORACLE and PostgreSQL use it even for SERIALIZABLE transactions!
  - But “serializable snapshot isolation” now in PostgreSQL
Snapshot Isolation Rules

• Each transaction receives a timestamp TS(T)

• Transaction T sees snapshot at time TS(T) of the database

• When T commits, updated pages are written to disk

• Write/write conflicts resolved by “first committer wins” rule
  – Loser gets aborted

• Read/write conflicts are ignored
Snapshot Isolation (Details)

• Multiversion concurrency control:
  – Versions of $X$: $X_{t1}$, $X_{t2}$, $X_{t3}$, . . .

• When T reads $X$, return $X_{TS(T)}$.

• When T writes $X$: if other transaction updated $X$, abort
  – Not faithful to “first committer” rule, because the other transaction $U$ might have committed after T. But once we abort T, U becomes the first committer 😊
What Works and What Not

• No dirty reads (Why ?)
• No inconsistent reads (Why ?)
  – A: Each transaction reads a consistent snapshot

• No lost updates ("first committer wins")

• Moreover: no reads are ever delayed

• However: read-write conflicts not caught !
Write Skew

T1:
READ(X);
if X >= 50
    then Y = -50; WRITE(Y)
COMMIT

T2:
READ(Y);
if Y >= 50
    then X = -50; WRITE(X)
COMMIT

In our notation:

\[ R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2 \]

Starting with X=50, Y=50, we end with X=-50, Y=-50. Non-serializable !!!
Questions/Discussions

• How does snapshot isolation (SI) compare to repeatable reads and serializable?
  – A: SI avoids most but not all phantoms (e.g., write skew)

• Note: Oracle & PostgreSQL implement it even for isolation level SERIALIZABLE
  – But most recently: “serializable snapshot isolation”

• How can we enforce serializability at the app level?
  – Recall that all read / write conflicts are ignored.
  – A: Use dummy writes for all reads to create write-write conflicts… but that is confusing for developers!!!
Commercial Systems

Always check documentation as DBMSs keep evolving and thus changing! Just to get an idea:

- **DB2**: Strict 2PL
- **SQL Server**:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- **PostgreSQL**: Multiversion concurrency control
- **Oracle**: Multiversion concurrency control
Important Lesson

• ACID transactions/serializability make it easy to develop applications
• BUT they add overhead and slow things down

• Lower levels of isolation reduce overhead
• BUT they are hard to reason about for developers!