3. Concurrency Control for Transactions

Part One

CSEP 545 Transaction Processing
Philip A. Bernstein

Copyright © 2005 Philip A. Bernstein

3.1 A Simple System Model

- Goal - Ensure serializable (SR) executions
- Implementation technique - Delay operations that would lead to non-SR results (e.g., set locks on shared data)
- For good performance minimize overhead and delay from synchronization operations
- First, we'll study how to get correct (SR) results
- Then, we'll study performance implications (mostly in Part Two)

Assumption - Atomic Operations

- We will synchronize Reads and Writes.
- We must therefore assume they're atomic - else we'd have to synchronize the finer-grained operations that implement Read and Write
- Read(x) - returns the current value of x in the DB
- Write(x, val) overwrites all of x (the whole page)
- This assumption of atomic operations is what allows us to abstract executions as sequences of reads and writes (without loss of information)
- What would w_i[x] r_j[x] mean?
- A loo, com m i t (c_i) and abort (a_i) are atomic

System Model

Transaction 1     Transaction 2     Transaction N
Start, Commit, Abort
Read(x), Write(x)

Data Manager

Database

3.2 Serializability Theory

- The theory is based on modeling executions as histories, such as
  \[ H_1 = r_1[x] r_2[x] w_1[x] c_i w_2[y] c_j \]
- First, characterize a concurrency control algorithm by the properties of histories it allows.
- Then prove that any history having these properties is SR
- Why bother? It helps you understand why concurrency control algorithms work.
Equivalence of Histories

- Two operations conflict if their execution order affects their return values or the DB state.
  - A read and write on the same data item conflict.
  - Two writes on the same data item conflict.
  - Two reads on the same data item do not conflict.

- Two histories are equivalent if they have the same operations and conflicting operations are in the same order in both histories.
  - Because only the relative order of conflicting operations can affect the result of the histories.

Examples of Equivalence

- The following histories are equivalent:
  - $H_1 = \text{r}_1[x] \text{r}_2[x] \text{w}_1[x] c_1 \text{w}_2[y] c_2$
  - $H_2 = \text{r}_2[x] \text{r}_1[x] \text{w}_1[x] c_1 \text{w}_2[y] c_2$
  - $H_3 = \text{r}_2[x] \text{r}_1[x] \text{w}_2[y] c_2 \text{w}_1[x] c_1$
  - $H_4 = \text{r}_2[x] \text{w}_2[y] c_2 \text{r}_1[x] \text{w}_1[x] c_1$

- But none of them are equivalent to:
  - $H_5 = \text{r}_1[x] \text{w}_1[x] \text{r}_2[x] c_1 \text{w}_2[y] c_2$

Serializable Histories

- A history is serializable if it is equivalent to a serial history.
- For example, $H_1 = \text{r}_1[x] \text{r}_2[x] \text{w}_1[x] c_1 \text{w}_2[y] c_2$
  - Is equivalent to $H_4 = \text{r}_2[x] \text{w}_2[y] c_2 \text{r}_1[x] \text{w}_1[x] c_1$
  - $r_2[x]$ and $w_1[x]$ are in the same order in $H_1$ and $H_4$.
- Therefore, $H_1$ is serializable.

Another Example

- $H_6 = \text{r}_1[x] \text{r}_2[x] \text{w}_1[x] \text{r}_3[x] \text{w}_2[y] \text{w}_3[x] c_3 \text{w}_1[y] c_1 c_2$
- Is equivalent to a serial execution of $T_2 T_1 T_3$.
- Each conflict implies a constraint on any equivalent serial history:
  - $H_6 = \text{r}_1[x] \text{r}_2[x] \text{w}_1[x] \text{r}_3[x] \text{w}_2[y] \text{w}_3[x] c_3 \text{w}_1[y] c_1 c_2$

Serialization Graphs

- A serialization graph, $SG(H)$, for history $H$ tells the effective execution order of transactions in $H$.
- Given history $H$, $SG(H)$ is a directed graph whose nodes are the committed transactions and whose edges are all $T_i fi T_k$ such that at least one of $T_i$’s operations precedes and conflicts with at least one of $T_k$’s operations.
- $H_6 = \text{r}_1[x] \text{r}_2[x] \text{w}_1[x] \text{r}_3[x] \text{w}_2[y] \text{w}_3[x] c_3 \text{w}_1[y] c_1 c_2$
  - $SG(H_6) = T_2fiT_1fiT_3$

The Serializability Theorem

- A history is SR if and only if $SG(H)$ is acyclic.
- Proof: (if) $SG(H)$ is acyclic. So let $H_s$ be a serial history consistent with $SG(H)$. Each pair of conflicting ops in $H$ induces an edge in $SG(H)$. Since conflicting ops in $H_s$ and $H$ are in the same order, $H_s \equiv H$, so $H$ is SR.
- (only if) $H$ is SR. Let $H_s$ be a serial history equivalent to $H$. We claim that if $T_i fi T_k$ in $SG(H)$, then $T_i$ precedes $T_k$ in $H_s$ (else $H_s \ne H$). If $SG(H)$ had a cycle, $T_i fi T_k fi ... fi T_k fi T_i$, then $T_i$ precedes $T_k$ in $H_s$, a contradiction. So $SG(H)$ is acyclic.
How to Use the Serializability Theorem

- Characterize the set of histories that a concurrency control algorithm allows
- Prove that any such history must have an acyclic serialization graph.
- Therefore, the algorithm guarantees SR executions.

We’ll use this soon to prove that locking produces serializable executions.

3.3 Synchronization Requirements for Recoverability

- In addition to guaranteeing serializability, synchronization is needed to implement aborts easily.
- When a transaction T aborts, the data manager rolls back all of T’s effects, including:
  - undoing T’s writes that were applied to the DB, and
  - aborting transactions that read values written by T
    (these are called cascading aborts)
- Example: w₁[k] x₂[k] w₂[y]
  - to abort T₁, we must undo w₁[k] and abort T₂
    (a cascading abort)

Recoverability

- If Tk reads from T₁ and T₁ aborts, then Tk must abort.
  - Example: w₁[k] r₂[x] a₁ implies T₂ must abort
- But what if Tk already committed? We’d be stuck.
  - Example: w₁[k] x₂[k] c₂ a₁
  - T₂ can’t abort after it commits
- Executions must be recoverable:
  A transaction T’s commit operation must follow the commit of every transaction from which T read.
  - Recoverable: w₁[k] x₂[k] c₂ a₁
  - Not recoverable: w₁[k] x₂[k] c₁ a₂
- Recoverability requires synchronizing operations.

Avoiding Cascading Aborts

- Cascading aborts are worth avoiding to
  - avoid complex bookkeeping, and
  - avoid an uncontrolled number of forced aborts
- To avoid cascading aborts, a data manager should ensure transactions only read committed data
- Example:
  - avoids cascading aborts: w₁[k] c₁ r₂[x]
  - allows cascading aborts: w₁[k] x₂[k] c₂ a₁
- A system that avoids cascading aborts also guarantees recoverability.

Strictness

- It’s convenient to undo a write, w[k], by restoring its before image (=the value of x before w[k] executed)
- Example: w₁[k,1] w₁[k,2] x₂[k,3] a₂
  - a₁ & a₂ can’t be in place by restoring before images
  - notice that w₁[k,2] w₂[k,3] a₁ a₂ w could be OK
- A system is strict if it only reads or overwrites committed data.

Strictness (cont’d)

- More precisely, a system is strict if it only executes ri[x] or wi[x] if all previous transactions that wrote x committed or aborted.
- Examples ("..." marks a non-strict prefix)
  - strict: w₁[k] c₁ w₂[k] a₁
  - not strict: w₁[k] w₂[k] ... c₁ a₁
  - strict: w₁[k] w₂[k] c₁ x₂[k] w₁[y] a₁
  - not strict: w₁[k] w₂[k] c₁ x₂[k] w₁[y] a₁
  - To see why strictness matters in the above histories, consider what happens if T₁ aborts
  - "Strict" implies "avoids cascading aborts,"
3.4 Two-Phase Locking

- Basic locking - Each transaction sets a lock on each data item before accessing the data
  - the lock is a reservation
  - there are read locks and write locks
  - if one transaction has a write lock on x, then no other transaction can have any lock on x

- Example
  - \text{rli}[x], \text{ru}[x], \text{wli}[x], \text{wu}[x] denote lock/unlock operations
  - \text{wl}[x] \text{w}[x] \text{rl}[x] \text{r}[x] is impossible
  - \text{wl}[x] \text{w}[x] \text{wu}[x] \text{ru}[x] \text{rl}[x] \text{r}[x] is OK

Basic Locking Isn’t Enough

- Basic locking doesn’t guarantee serializability
  - \text{r}[x] \text{r}[y] \text{w}[x] \text{wu}[y] \text{ri}[y] \text{wu}[x] \text{ci}
  - Eliminating the lock operations, we have \text{r}[x] \text{r}[y] \text{wu}[x] \text{wu}[y] \text{ci}
  - The problem is that locks aren’t being released properly.

Two-Phase Locking (2PL) Protocol

- A transaction is two-phase locked if:
  - before reading x, it sets a read lock on x
  - before writing x, it sets a write lock on x
  - it holds each lock until after it executes the corresponding operation
  - after its first unlock operation, it requests no new locks

- Each transaction sets locks during a growing phase and releases them during a shrinking phase.

- Example - on the previous page \text{T}_2 is two-phase locked, but \text{T}_1 since \text{ru}[x] < \text{wl}[y]
  - use \text{“<”} for “precedes”

2PL and Recoverability

- 2PL does not guarantee recoverability
- This non-recoverable execution is 2-phase locked
  - \text{w}[x] \text{w}[y] \text{wu}[x] \text{ri}[y] \text{wu}[y] \text{ci} \ldots \text{ci}
  - hence, it is not strict and allows cascading aborts
- However, holding write locks until after commit or abort guarantees strictness
  - and hence avoids cascading aborts and is recoverable
- In the above example, \text{T}_1 is not recoverable before its first unlock
  - the unlock operation \text{wu}[x] \text{wu}[y] \text{ci} \text{wu}[x] \text{ri}[y] \text{ci} \text{wu}[y] \text{ci}

2PL Theorem: If all transactions in an execution are two-phase locked, then the execution is SR.

Proof: Define \text{T}_i \Rightarrow \text{T}_k if either
  - \text{T}_i reads x and \text{T}_k later writes x, or
  - \text{T}_i writes x and \text{T}_k later reads or writes x

- If \text{T}_i \Rightarrow \text{T}_k, then \text{T}_i released a lock before \text{T}_k obtained some lock.
- If \text{T}_i \Rightarrow \text{T}_k \Rightarrow \text{T}_m, then \text{T}_i released a lock before \text{T}_m obtained some lock (because \text{T}_k is two-phase).
- If \text{T}_i \Rightarrow \text{T}_k \Rightarrow \ldots \Rightarrow \text{T}_n, then \text{T}_i released a lock before \text{T}_n obtained some lock, breaking the 2-phase rule.
- So there cannot be a cycle. By the Serializability Theorem, the execution is SR.

Automating Locking

- 2PL can be hidden from the application
  - When a data manager gets a \text{read or write} operation from a transaction, it sets a read or write lock.
  - How does the data manager know it’s safe to release locks (and be two-phase)?
- Ordinarily, the data manager holds a transaction’s locks until it commits or aborts. A data manager
  - can release read locks after it receives commit
  - releases write locks only after processing commit, to ensure strictness
3.5 Preserving Transaction Handshakes

- Read and Write are the only operations the system will control to attain serializability.
- So, if transactions communicate via messages, then implement SendMsg as Write, and ReceiveMsg as Read.
- Else, you could have the following:
  \[ w_1[x] r_2[x] send_2[M] receive_1[M] \]
  - data manager didn’t know about send/receive and thought the execution was SR.
- Also watch out for brain transport.

Brain Transport (cont’d)

- For practical purposes, if user waits for \( T_1 \) to commit before starting \( T_2 \), then the data manager can ignore brain transport.
- This is called a transaction handshake \( (T_1 \) commits before \( T_2 \) starts).
- Reason - Locking preserves the order imposed by transaction handshakes
  - e.g., it serializes \( T_1 \) before \( T_2 \).

2PL Preserves Transaction Handshakes

- 2PL serializes transactions (abbr. txns) consistent with all transaction handshakes. I.e., there’s an equivalent serial execution that preserves the transaction order of transaction handshakes.
- This isn’t true for arbitrary SR executions. E.g.,
  \[ r_1[x] w_2[x] c_2 \]
  - \( T_2 \) commits before \( T_3 \) starts, but the only equivalent serial execution is \( T_3 T_1 T_2 \).
  \[ r_1[x] r_1[x] w_1[y] w_1[y] w_2[x] w_2[x] w_1[y] c_2 \]
  but now we’re stuck, since we can’t set \( rl_3[y] \).
  - So the history cannot occur using 2PL.

2PL Preserves Transaction Handshakes (cont’d)

- Stating this more formally...
- Theorem:
  \[ \text{For any 2PL execution } H, \]
  \[ \text{there is an equivalent serial execution } H_s, \]
  such that for all \( T_i, T_k \),
  \[ \text{if } T_i \text{ commits before } T_k \text{ started in } H, \]
  \[ \text{then } T_i \text{ precedes } T_k \text{ in } H_s. \]

Brain Transport One Last Time

- If a user reads committed displayed output of \( T_i \)
  and uses that displayed output as input to transaction \( T_k \), then he/she should wait for \( T_i \) to commit before starting \( T_k \).
- The user can then rely on transaction handshake preservation to ensure \( T_i \) is serialized before \( T_k \).

Transactions Can Communicate via Brain Transport

\[ T_1: \text{Start} \]
\[ \ldots \]
\[ \text{Display output Commit} \]
\[ \text{Brain transport} \]

\[ T_2: \text{Start} \]
\[ \ldots \]
\[ \text{Get input from display Commit} \]
3.6 Implementing Two-Phase Locking

- Even if you never implement a DB system, it's valuable to understand locking in implementation, because it can have a big effect on performance.
- A data manager implements locking by:
  - implementing a lock manager
  - setting a lock for each Read and Write
  - handling deadlocks

How to Implement SQL

- Query Optimizer - translates SQL into an ordered expression of relational DB operators (Select, Project, Join)
- Query Executor - executes the ordered expression by running a program for each operator, which in turn accesses records of files
- A cross method - provides indexed record-at-a-time access to files (OpenScan, GetNext, ...)
- Page-oriented files - Read or Write (page address)

Which Operations Get Synchronized?

- Record-oriented operations
- Page-oriented operations

Lock Manager

- A lock manager services the operations:
  - Lock (trans-id, data-item-id, mode)
  - Unlock (trans-id, data-item-id)
  - Unlock (trans-id)


<table>
<thead>
<tr>
<th>Data Item</th>
<th>List of Locks</th>
<th>Wait List</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>[T1, r] [T3, w]</td>
<td>[T2, w]</td>
</tr>
<tr>
<td>y</td>
<td>[T4, w]</td>
<td>[T5, w] [T6, r]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lock Manager (cont’d)

- In MS SQL Server:
  - Locks are approx 32 bytes each.
  - Each lock contains a Database-ID, Object-ID, and other resource-specific lock information such as record ID (RID) or key.
  - Each lock is attached to lock resource block (64 bytes) and lock owner block (32 bytes).

Locking Granularity

- Granularity - size of data items to lock
  - e.g., files, pages, records, fields
- Coarse granularity in pages
  - very few locks, so little locking overhead
  - must lock large chunks of data, so high chance of conflict, so concurrency may be low
- Fine granularity in pages
  - many locks, so high locking overhead
  - locking conflict occurs only when two transactions try to access the exact same data concurrently
- High performance TP requires record locking

Multigranularity Locking (MGL)

- A low different base to lock at different granularity
  - big queries should lock coarse-grained data (e.g., tables)
  - short transactions lock fine-grained data (e.g., row)
- Lock manager can’t detect these conflicts
  - each data item (e.g., table or row) has a different id
  - Multigranularity locking “trick”
  - exploit the natural hierarchy of data containment
  - before locking fine-grained data, set intention locks on coarse-grained data that contains it:
    - e.g., before setting a read-lock on a row, get an intention-read-lock on the table that contains the row
  - intention-read locks conflict with a write lock

3.7 Deadlocks

- A set of transactions is deadlocked if every transaction in the set is blocked and will remain blocked unless the system intervenes.
  - Example: r1[x] granted, r2[y] granted, w1[x] blocked, w2[y] blocked and deadlocked
- Deadlock is 2PL’s way to avoid non-SR executions
  - r1[x] r1[x] r1[y] r2[y]... can’t run w1[x] w2[y] and be SR
- To repair a deadlock, you must abort a transaction
  - if you released a transaction’s lock without aborting it, you’d break 2PL

Deadlock Prevention

- Never grant a lock that can lead to deadlock
- Useless for TP, because it would require running transactions serially.
  - Example: to prevent the previous deadlock, r1[x] r1[y] w1[x] w1[y], the system can’t grant r1[y]
- A voicing deadlock by resource ordering is unusable in general, since it overly constrains applications.
  - But may help for certain high frequency deadlocks
- Setting all locks when txn begins requires too much advance knowledge and reduces concurrency.

Deadlock Detection

- Detection approach: Detect deadlocks automatically, and abort a deadlocked transaction (the victim).
  - It’s the preferred approach, because it:
    - allows higher resource utilization and
    - uses cheaper algorithms
- Tim out-based deadlock detection - If a transaction is blocked for too long, then abort it:
  - Simple and easy to implement
  - But aborts unnecessarily and
  - Some deadlocks persist for too long
Detection Using Waits-For Graph

- Explicit deadlock detection
  - Use a Waits-For Graph
  - Nodes = \{transactions\}
  - Edges = \{Ti \backslash fi \ Tk | Ti is waiting for Tk to release a lock\}
- Example (previous deadlock)

- Theorem: If there’s a deadlock, then the waits-for graph has a cycle.

Cyclic Restart

- Transactions can cause each other to abort forever.
  - T_1 starts running. Then T_2 starts running.
  - They deadlock and T_1 (the oldest) is aborted.
  - T_2 restarts, bumps into T_1 and again deadlocks
  - T_2 (the oldest) is aborted ...
- Choosing the youngest in a cycle as victim avoids cyclic restart, since the oldest running transaction is never the victim.
- Can combine with other heuristics, e.g., fewest-locks

MS SQL Server

- Aborts the transaction that is "cheapest" to rollback.
  - "Cheapest" is determined by the amount of log generated.
  - Allows transactions that you’ve invested a lot in to complete.
- SET DEADLOCK_PRIORITY LOW (vs. NORMAL) causes a transaction to sacrifice itself as a victim

Distributed Locking

- Suppose a transaction can access data at any data manager
- Each data manager sets locks in the usual way
- When a transaction commits or aborts, it runs two-phase commit to notify all data managers it accessed
- The only remaining issue is distributed deadlock

Distributed Deadlock

- The deadlock spans two nodes. Neither node alone can see it.

Node 1

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>[k] (blocked)</td>
</tr>
</tbody>
</table>
```

Node 2

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[y] (blocked)</td>
</tr>
</tbody>
</table>
```

- Timeout-based detection is popular. Its weaknesses are less prominent in the distributed case:
  - aborts unnecessarily and some deadlocks persist too long
  - possibly about younger unblocked transaction to avoid cyclic restart
Oracle Deadlock Handling

- Uses a waits-for graph for single-server deadlock detection.
- The transaction that detects the deadlock is the victim.
- Uses timeouts to detect distributed deadlocks.

Fancier Distributed Deadlock Detection

- Use waits-for graph cycle detection with a central deadlock detection server.
  - More work than timestamp-based detection, and no evidence it does better, performance-wise.
  - Phantom deadlocks? No, because each waits-for edge is an SG edge. So, WFG cycle => SG cycle (modulo spontaneous aborts).
- Path pushing (a.k.a. flooding): Send paths Ti ... fi Tk to each node where Tk might be blocked.
  - Detects short cycles quickly.
  - Hard to know where to send paths.
  - Possibly too many messages.

What's Coming in Part Two?

- Locking Performance
- A more detailed look at multigranularity locking
- Hot Spot Techniques
- Query-Update Techniques
- Phantom S
- B-Trees and Tree Locking

Locking Performance

- The following is oversimplified. We'll revisit it.
- Deadlocks are rare.
  - Typically 1-2% of transactions deadlock.
- Locking performance problems are not rare.
- The problem is too much blocking.
- The solution is to reduce the "locking load".
- Good heuristic: If more than 30% of transactions are blocked, then reduce the number of concurrent transactions.