8. Concurrency Control for Transactions

Part Two

CSEP 545 Transaction Processing

Philip A. Bernstein

Copyright ©2007 Philip A. Bernstein
Outline

1. A Model for Concurrency Control
2. Serializability Theory
3. Synchronization Requirements for Recoverability
4. Two-Phase Locking
5. Implementing Two-Phase Locking
6. Locking Performance
7. Multigranularity Locking (revisited)
8. Hot Spot Techniques
9. Query-Update Techniques
10. Phantoms
11. Shared Disk Systems
12. B-Trees
13. Tree locking
8.6 Locking Performance

- Deadlocks are rare
  - up to 1% - 2% of transactions deadlock

- The one exception to this is lock conversions
  - r-lock a record and later upgrade to w-lock
  - e.g., $T_i = \text{read}(x) \ldots \text{write}(x)$
  - if two txns do this concurrently, they’ll deadlock
    (both get an r-lock on $x$ before either gets a w-lock)
  - To avoid lock conversion deadlocks, get a w-lock first
    and down-grade to an r-lock if you don’t need to write.
  - Use SQL Update statement or explicit program hints
Conversions in MS SQL Server

• Update-lock prevents lock conversion deadlock.
  – Conflicts with other update and write locks, but not with read locks.
  – Only on pages and rows (not tables)

• You get an update lock by using the UPDLOCK hint in the FROM clause

```sql
Select Foo.A
From Foo (UPDLOCK)
Where Foo.B = 7
```
Blocking and Lock Thrashing

- The locking performance problem is too much delay due to blocking
  - little delay until locks are saturated
  - then major delay, due to the locking bottleneck
  - **thrashing** - the point where throughput decreases with increasing load
More on Thrashing

• It’s purely a blocking problem
  – It happens even when the abort rate is low

• As number of transactions increase
  – each additional transaction is more likely to block
  – but first, it gathers some locks, increasing the probability others will block (negative feedback)
Avoiding Thrashing

• If over 30% of active transactions are blocked, then the system is (nearly) thrashing so reduce the number of active transactions

• Timeout-based deadlock detection mistakes
  – They happen due to long lock delays
  – So the system is probably close to thrashing
  – So if deadlock detection rate is too high (over 2%) reduce the number of active transactions
Interesting Sidelights

• By getting all locks before transaction Start, you can increase throughput at the thrashing point because blocked transactions hold no locks
  – But it assumes you get exactly the locks you need and retries of get-all-locks are cheap

• Pure restart policy - abort when there’s a conflict and restart when the conflict disappears
  – If aborts are cheap and there’s low contention for other resources, then this policy produces higher throughput before thrashing than a blocking policy
  – But response time is greater than a blocking policy
How to Reduce Lock Contention

• If each transaction holds a lock \( L \) for \( t \) seconds, then the maximum throughput is \( 1/t \) txns/second

Start \[ \text{Lock } L \text{ Commit} \]

• To increase throughput, reduce \( t \) (lock holding time)
  – Set the lock later in the transaction’s execution (e.g., defer updates till commit time)
  – Reduce transaction execution time (reduce path length, read from disk before setting locks)
  – Split a transaction into smaller transactions
Reducing Lock Contention (cont’d)

• Reduce number of conflicts
  – Use finer grained locks, e.g., by partitioning tables vertically

<table>
<thead>
<tr>
<th>Part#</th>
<th>Price</th>
<th>OnHand</th>
<th>PartName</th>
<th>CatalogPage</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Part#</th>
<th>Price</th>
<th>OnHand</th>
<th>PartName</th>
<th>CatalogPage</th>
</tr>
</thead>
</table>

  – Use record-level locking (i.e., select a database system that supports it)
Mathematical Model of Locking

- K locks per transaction
- D lockable data items
- N transactions each own K/2 locks on average
  - KN/2 in total
- Each lock request has probability KN/2D of conflicting with an existing lock.
- Each transaction requests K locks, so its probability of experiencing a conflict is K²N/2D.
- Probability of a deadlock is proportional to K⁴N/D²
  - Prob(deadlock) / Prop(conflict) = K²/D
  - if K=10 and D = 10^6, then K²/D = .0001
8.7 Multigranularity Locking (MGL)

- Allow different txns to lock at different granularity
  - big queries should lock coarse-grained data (e.g. tables)
  - short transactions lock fine-grained data (e.g. rows)
- 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
Before setting a read lock on R2.3, first set an intention-read lock on DB1, then A2, and then F2.

# MGL Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>w</th>
<th>ir</th>
<th>iw</th>
<th>riw</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>w</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>ir</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>iw</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>riw</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

riw = read with intent to write, for a scan that updates some of the records it reads

- E.g., ir conflicts with w because ir says there’s a fine-grained r-lock that conflicts with a w-lock on the container.
- To r-lock an item, need an r-, ir- or riw-lock on its parent.
- To w-lock an item, need a w-, iw- or riw-lock on its parent.
MGL Complexities

- Relational DBMSs use MGL to lock SQL queries, short updates, and scans with updates.
- Use lock escalation - start locking at fine-grain and escalate to coarse grain after $n^{th}$ lock is set.
- The lock type graph is a directed acyclic graph, not a tree, to cope with indices.
- R-lock one path to an item. W-lock all paths to it.
MS SQL Server

- MS SQL Server can lock at table, page, and row level.
- Uses intention read (“share”) and intention write (“exclusive”) locks at the table and page level.
- Tries to avoid escalation by choosing the “appropriate” granularity when the scan is instantiated.
8.8 Hot Spot Techniques

- If eachtxn holds a lock for \( t \) seconds, then the max throughput is \( 1/t \) txns/second for that lock.
- Hot spot - A data item that’s more popular than others, so a large fraction of active txns need it
  - Summary information (total inventory)
  - End-of-file marker in data entry application
  - Counter used for assigning serial numbers
- Hot spots often create a convoy of transactions. The hot spot lock serializes transactions.
Hot Spot Techniques (cont’d)

• Special techniques are needed to reduce $t$
  – Keep the hot data in main memory
  – Delay operations on hot data till commit time
  – Use optimistic methods
  – Batch up operations to hot spot data
  – Partition hot spot data
Delaying Operations Until Commit

- Data manager logs each transaction’s updates
- Only applies the updates (and sets locks) after receiving Commit from the transaction
- IMS Fast Path uses this for
  - Data Entry DB
  - Main Storage DB
- Works for write, insert, and delete, but not read
Locking Higher-Level Operations

- Read is often part of a read-write pair, such as `Increment(x, n)`, which adds constant `n` to `x`, but doesn’t return a value.
- Increment (and Decrement) commute
- So, introduce Increment and Decrement locks

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>w</th>
<th>inc</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>w</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>inc</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>dec</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

- But if Inc and Dec have a threshold (e.g. a quantity of zero), then they conflict (when the threshold is near)
Solving the Threshold Problem
Another IMS Fast Path Technique

- Use a blind Decrement (no threshold) and Verify(x, n), which returns true if $x \geq n$
- Re-execute Verify at commit time
  - If it returns a different value than it did during normal execution, then abort
  - It’s like checking that the threshold lock you didn’t set during Decrement is still valid.

```c
bEnough = Verify(iQuantity, n);
If (bEnough) Decrement(iQuantity, n)
else print ("not enough");
```
Optimistic Concurrency Control

• The Verify trick is optimistic concurrency control
• Main idea - execute operations on shared data without setting locks. At commit time, test if there were conflicts on the locks (that you didn’t set).
• Often used in client/server systems
  – Client does all updates in cache without shared locks
  – At commit time, try to get locks and perform updates
Batching

- Transactions add updates to a mini-batch and only periodically apply the mini-batch to shared data.
  - Each process has a private data entry file, in addition to a global shared data entry file
  - Each transaction appends to its process’ file
  - Periodically append the process file to the shared file

- Tricky failure handling
  - Gathering up private files
  - Avoiding holes in serial number order
Partitioning

• Split up inventory into partitions
• Each transaction only accesses one partition
• Example
  – Each ticket agency has a subset of the tickets
  – If one agency sells out early, it needs a way to get more tickets from other agencies (partitions)
8.9 Query-Update Techniques

• Queries run for a long time and lock a lot of data — a performance nightmare when trying also to run short update transactions

• There are several good solutions
  – Use a data warehouse
  – Accept weaker consistency guarantees
  – Use multiversion data

• Solutions trade data quality or timeliness for performance
Data Warehouse

• A data warehouse contains a snapshot of the DB which is periodically refreshed from the TP DB
• All queries run on the data warehouse
• All update transactions run on the TP DB
• Queries don’t get absolutely up-to-date data
• How to refresh the data warehouse?
  – Stop processing transactions and copy the TP DB to the data warehouse. Possibly run queries while refreshing
  – Treat the warehouse as a DB replica and use a replication technique
Degrees of Isolation

- Serializability = *Degree 3 Isolation*
- Degree 2 Isolation (a.k.a. cursor stability)
  - Data manager holds read-lock(x) only while reading x, but holds write locks till commit (as in 2PL)
  - E.g. when scanning records in a file, each get-next-record releases lock on current record and gets lock on next one
  - read(x) is not “repeatable” within a transaction, e.g.,
    \[
    \text{rl}_1[x] \ r_1[x] \text{ru}_1[x] \ \text{wl}_2[x] \ w_2[x] \text{wu}_2[x] \text{rl}_1[x] \ r_1[x] \text{ru}_1[x]
    \]
  - Degree 2 is commonly used by ISAM file systems
  - Degree 2 is often a DB system’s default behavior!
  And customers seem to accept it!!!
Degrees of Isolation (cont’d)

• Could run queries Degree 2 and updaters Degree 3  
  – Updaters are still serializable w.r.t. each other
• Degree 1 - no read locks; hold write locks to commit
• Unfortunately, SQL concurrency control standards have been stated in terms of “repeatable reads” and “cursor stability” instead of serializability, leading to much confusion.
ANSI SQL Isolation Levels

- Uncommitted Read - Degree 1
- Committed Read - Degree 2
- Repeatable Read - Uses read locks and write locks, but allows “phantoms”
- Serializable - Degree 3
• Lock hints in SQL FROM clause
  – All the ANSI isolation levels, plus …
  – UPDLOCK - use update locks instead of read locks
  – READPAST - ignore locked rows (if running read committed)
  – PAGLOCK - use page lock when the system would otherwise use a table lock
  – TABLOCK - shared table lock till end of command or transaction
  – TABLOCKX - exclusive table lock till end of command or transaction
Multiversion Data

- Assume record granularity locking
- Each write operation creates a new version instead of overwriting existing value.
- So each logical record has a sequence of versions.
- Tag each record with transaction id of the transaction that wrote that version

<table>
<thead>
<tr>
<th>Tid</th>
<th>Previous</th>
<th>E#</th>
<th>Name</th>
<th>Other fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>null</td>
<td>1</td>
<td>Bill</td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>123</td>
<td>1</td>
<td>Bill</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>null</td>
<td>2</td>
<td>Sue</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>134</td>
<td>2</td>
<td>Sue</td>
<td></td>
</tr>
<tr>
<td>227</td>
<td>null</td>
<td>27</td>
<td>Steve</td>
<td></td>
</tr>
</tbody>
</table>
Multiversion Data (cont’d)

- Execute update transactions using ordinary 2PL
- Execute queries in *snapshot mode*
  - System keeps a *commit list* of tids of all committed txns
  - When a query starts executing, it reads the commit list
  - When a query reads x, it reads the latest version of x written by a transaction on its commit list
  - Thus, it reads the database state that existed when it started running
Commit List Management

• Maintain and periodically recompute a tid $T_{Oldest}$, such that
  – Every active txn’s tid is greater than $T_{Oldest}$
  – Every new tid is greater than $T_{Oldest}$
  – For every committed transaction with tid $\leq T_{Oldest}$, its versions are committed
  – For every aborted transaction with tid $\leq T_{Oldest}$, its versions are wiped out

• Queries don’t need to know tids $\leq T_{Oldest}$
  – So only maintain the commit list for tids $> T_{Oldest}$
Multiversion Garbage Collection

• Can delete an old version of x if no query will ever read it
  – There’s a later version of x whose tid \( \leq \) T-Oldest (or is on every active query’s commit list)

• Originally used in Prime Computer’s CODASYL DB system and Oracle’s Rdb/VMS
Oracle Multiversion Concurrency Control

• Data page contains latest version of each record, which points to older version in rollback segment.
• Read-committed query reads data as of its start time.
• Read-only isolation reads data as of transaction start time.
• “Serializable” txn reads data as of the txn’s start time.
  – An update checks that the updated record was not modified after txn start time.
  – If that check fails, Oracle returns an error.
  – If there isn’t enough history for Oracle to perform the check, Oracle returns an error. (You can control the history area’s size.)
  – What if $T_1$ and $T_2$ modify each other’s readset concurrently?
Oracle Concurrency Control (cont’d)

\[ r_1[x] \ r_1[y] \ r_2[x] \ r_2[y] \ w_1[x'] \ c_1 \ w_2[y'] \ c_2 \]

- The result is not serializable!
- In any SR execution, one transaction would have read the other’s output
### 8.10 Phantoms

- Problems when using 2PL with inserts and deletes

**Accounts**

<table>
<thead>
<tr>
<th>Acct#</th>
<th>Location</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Seattle</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>Tacoma</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>Tacoma</td>
<td>300</td>
</tr>
</tbody>
</table>

**Assets**

<table>
<thead>
<tr>
<th>Location</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>400</td>
</tr>
<tr>
<td>Tacoma</td>
<td>500</td>
</tr>
</tbody>
</table>

\[ T_1: \text{Read Accounts 1, 2, and 3} \]
\[ T_2: \text{Insert Accounts}[4, \text{Tacoma}, 100] \]
\[ T_2: \text{Read Assets(Tacoma), returns 500} \]
\[ T_2: \text{Write Assets(Tacoma, 600)} \]
\[ T_1: \text{Read Assets(Tacoma), returns 600} \]
\[ T_1: \text{Commit} \]

The phantom record
The Phantom Phantom Problem

• It looks like $T_1$ should lock record 4, which isn’t there!

• Which of $T_1$’s operations determined that there were only 3 records?
  – Read end-of-file?
  – Read record counter?
  – SQL Select operation?

• This operation conflicts with $T_2$’s Insert Accounts[4,Tacoma,100]

• Therefore, Insert Accounts[4,Tacoma,100] shouldn’t run until after $T_1$ commits
Avoiding Phantoms - Predicate Locks

• Suppose a query reads all records satisfying predicate P. For example,
  – Select * From Accounts Where Location = “Tacoma”
  – Normally would hash each record id to an integer lock id
  – And lock control structures. Too coarse grained.

• Ideally, set a read lock on P
  – which conflicts with a write lock Q if some record can satisfy (P and Q)

• For arbitrary predicates, this is too slow to check
  – Not within a few hundred instructions, anyway
Precision Locks

- Suppose update operations are on single records
- Maintain a list of predicate Read-locks
- Insert, Delete, & Update write-lock the record and check for conflict with all predicate locks
- Query sets a read lock on the predicate and check for conflict with all record locks
- Cheaper than predicate satisfiability, but still too expensive for practical implementation.
8.12 B-Trees

• An *index* maps field values to record ids.
  – Record id = [page-id, offset-within-page]
  – Most common DB index structures: hashing and B-trees
  – DB index structures are *page-oriented*

• Hashing uses a function $H: V \rightarrow B$, from field values to block numbers.
  – $V =$ social security numbers. $B = \{1 .. 1000\}$
    $H(v) = v \mod 1000$
  – If a page overflows, then use an extra overflow page
  – At 90% load on pages, 1.2 block accesses per request!
  – BUT, doesn’t help for key range access ($10 < v < 75$)
B-Tree Structure

- Index node is a sequence of [pointer, key] pairs
- \( K_1 < K_2 < \ldots < K_{n-2} < K_{n-1} \)
- \( P_1 \) points to a node containing keys \(< K_1 \)
- \( P_i \) points to a node containing keys in range \([K_{i-1}, K_i)\)
- \( P_n \) points to a node containing keys \( > K_{n-1} \)
- So, \( K_1 < K_2 < \ldots < K_{n-2} < K_{n-1} \)
Example $n=3$

- Notice that leaves are sorted by key, left-to-right
- Search for value $v$ by following path from the root
- If key = 8 bytes, ptr = 2 bytes, page = 4K, then $n = 409$
- So 3-level index has up to 68M leaves ($409^3$)
- At 20 records per leaf, that’s 136M records
Insertion

- To insert key \( v \), search for the leaf where \( v \) should appear.
- If there’s space on the leaf, insert the record.
- If no, split the leaf in half, and split the key range in its parent to point to the two leaves.

To insert key 15:
- split the leaf
- split the parent’s range \([0, 19)\) to \([0, 15)\) and \([15, 19)\)
- if the parent was full, you’d split that too (not shown here)
- this automatically keeps the tree balanced
B-Tree Observations

- Delete algorithm merges adjacent nodes < 50% full, but rarely used in practice
- Root and most level-1 nodes are cached, to reduce disk accesses
- Secondary (non-clustered) index - Leaves contain [key, record id] pairs.
- Primary (clustered) index - Leaves contain records
- Use key prefix for long (string) key values
  - drop prefix and add to suffix as you move down the tree
Key Range Locks

- Lock on B-tree key range is a cheap predicate lock

- Select Dept Where ((Budget > 250) and (Budget < 350))
- lock the key range [221, 352) record
- only useful when query is on an indexed field

- Commonly used with multi-granularity locking
  - Insert/delete locks record and intention-write locks range
  - MGL tree defines a fixed set of predicates, and thereby avoids predicate satisfiability
8.13 Tree Locking

- Can beat 2PL by exploiting root-to-leaf access in a tree
- If searching for a leaf, after setting a lock on a node, release the lock on its parent

wl(A) wl(B) wu(A) wl(E) wu(B)

- The lock order on the root serializes access to other nodes
B-tree Locking

- Root lock on a B-tree is a bottleneck
- Use tree locking to relieve it
- Problem: node splits

If you unlock P before splitting C, then you have to back up and lock P again, which breaks the tree locking protocol.

- So, don’t unlock a node till you’re sure its child won’t split (i.e. has space for an insert)
- Implies different locking rules for different ops (search vs. insert/update)
B-link Optimization

- B-link tree - Each node has a side pointer to the next
- After searching a node, you can release its lock before locking its child
  - $r_1[P] r_2[P] r_2[C] w_2[C] w_2[C'] w_2[P] r_1[C] r_1[C']$

- Searching has the same behavior as if it locked the child before releasing the parent … and ran later (after the insert)