5. Concurrency Control for Transactions

CSE 593 Transaction Processing
Philip A. Bernstein
Copyright ©2001 Philip A. Bernstein

5.1 A Model for Concurrency Control

The Problem

• 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

System Model

Transaction 1
Start, SQL Ops
Commit, Abort

Database

Query Optimizer
Query Executor
Access Method
(record-oriented files)
Page-oriented Files

Database Files

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
• Access methods - provides indexed record-at-a-time access to files (OpenScan, GetNext, …)
• Page-oriented files - Read or Write (page address)

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. Hot Spot Techniques
8. Query-Update Techniques
9. Phantoms
10. B-Trees
11. Tree locking

Which Operations Get Synchronized?

Query Optimizer
Query Executor
Access Method
(record-oriented files)
Page-oriented Files

SQL operations
Record-oriented operations
Page-oriented operations

• It’s a tradeoff between
  – amount of concurrency and
  – overhead and complexity of synchronization
• For now, assume page operations
  – notation: r[x], w[x] where “x” is a page and use the neutral term data manager
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).
  - Otherwise, what would \( w_i[x] r_i[x] \) mean?

Assumption - Txns communicate only via Read and Write
- 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:
  \( w_i[x] r_i[x] send_i[M] receive_i[M] \)
  - data manager didn’t know about send/receive and
    thought the execution was SR.
- Also watch out for brain transport

Transactions Can Communicate via 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 \( T_2 \).
- Stating this precisely and proving it is non-trivial.
- … more later ….

5.2 Serializability Theory
- The theory is based on modeling executions as
  histories, such as
  \[
  H_1 = r_1[x] r_2[x] w_1[x] c_1 w_2[y] c_2
  \]
- 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
  \[ H_1 = r_1[x] r_2[x] w_1[x] c_1 w_2[y] c_2 \]
  \[ H_2 = r_2[x] r_1[x] w_1[x] c_1 w_2[y] c_2 \]
  \[ H_3 = r_2[x] w_2[y] c_2 r_1[x] w_1[x] c_1 \]
  \[ H_4 = r_2[x] w_2[y] c_2 r_1[x] w_1[x] c_1 \]

- But none of them are equivalent to
  \[ H_5 = r_1[x] w_1[x] r_2[x] c_1 w_2[y] c_2 \]
  because \( r_2[x] \) and \( w_1[x] \) conflict and \( r_2[x] \) precedes \( w_1[x] \) in \( H_1 - H_4 \), but \( w_1[x] \) precedes \( r_2[x] \) in \( H_5 \).

Serializable Histories

- A history is serializable if it is equivalent to a serial history
- For example,
  \[ H_1 = r_1[x] r_2[x] w_1[x] c_1 w_2[y] c_2 \]
  is equivalent to
  \[ H_4 = r_2[x] w_2[y] c_2 r_1[x] w_1[x] c_1 \]
  (\( r_2[x] \) and \( w_1[x] \) are in the same order in \( H_1 \) and \( H_4 \).
- Therefore, \( H_1 \) is serializable.

Another Example

- \( H_6 = r_1[x] r_2[x] w_1[x] r_3[x] w_2[y] w_3[x] c_3 w_1[y] c_1 c_2 \)
  is equivalent to a serial execution of \( T_2 T_1 T_3 \),
  \[ H_7 = r_2[x] w_2[y] c_2 r_1[x] w_1[x] w_1[y] c_1 r_3[x] w_3[x] c_3 \]
- Each conflict implies a constraint on any equivalent serial history:
  \[ H_6 = r_1[x] r_2[x] w_1[x] r_3[x] w_2[y] w_3[x] c_3 w_1[y] c_1 c_2 \]

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 \( T_i \rightarrow T_k \) such that at least one of \( T_i \)'s operations precedes and conflicts with at least one of \( T_k \)'s operations
  \[ H_6 = r_1[x] r_2[x] w_1[x] r_3[x] w_2[y] w_3[x] c_3 w_1[y] c_1 c_2 \]
  \[ SG(H_6) = T_2 \rightarrow T_1 \rightarrow T_3 \]

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 \( H \). Claim that if \( T_1 \rightarrow T_k \) in \( SG(H) \), then \( T_i \) precedes \( T_k \) in \( H_s \) (else \( H_s \not\equiv H \)). If \( SG(H) \) had a cycle, \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \), then \( T_1 \) precedes \( T_1 \) 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.
5.3 Synchronization Requirements for Recoverability

- In addition to guaranteeing serializability, synchronization is needed to implement abort easily.
- When a transaction $T$ aborts, the data manager wipes out 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 - $w_1[x] r_2[x] w_2[y]$
  - to abort $T_1$, we must undo $w_1[x]$ and abort $T_2$ (a cascading abort)

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: $w_1[x] c_1 r_2[x]$
  - allows cascading aborts: $w_1[x] r_2[x] a_1$
- A system that avoids cascading aborts also guarantees recoverability.

Strictness

- More precisely, a system is strict if it only executes $r_i[x]$ or $w_i[x]$ if all previous transactions that wrote $x$ committed or aborted.
- Examples (“…” marks a non-strict prefix)
  - strict: $w_1[x] c_1 w_2[x] a_1$
  - not strict: $w_1[x] w_2[x] … a_1 a_2$
  - strict: $w_1[x] w_3[y] c_1 w_2[y] r_2[x] a_1$
  - not strict: $w_1[x] w_3[y] w_2[y] a_2 r_2[x] a_2$
- “Strict” implies “avoids cascading aborts.”

5.4 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
  - $r_1[x], r_2[x], w_1[x], w_2[x]$ denote lock/unlock operations
  - $w_1[x] w_1[x] r_2[x] r_2[x]$ is impossible
  - $w_1[x] w_1[x] r_2[x] r_2[x]$ is OK
Basic Locking Isn’t Enough
- Basic locking doesn’t guarantee serializability
- Eliminating the lock operations, we have r1[x] r2[y] w2[x] c2 which isn’t SR
- 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 T2 is two-phase locked, but not T1 since ru1[x] < wl1[y]
  - use “<” for “precedes”

2PL Theorem: If all transactions in an execution are two-phase locked, then the execution is SR.
Proof: Define Ti ⇒ Tk if either
  - Ti read x and Tk later wrote x, or
  - Ti wrote x and Tk later read or wrote x
- If Ti ⇒ Tk, then Ti released a lock before Tk obtained some lock.
- If Ti ⇒ Tk ⇒ Tm, then Ti released a lock before Tm obtained some lock (because Tk is two-phase).
- If Ti ⇒ ... ⇒ Ti, then Ti released a lock before Ti obtained some lock, breaking the 2-phase rule.
- So there cannot be a cycle. By the Serializability Theorem, the execution is SR.

2PL and Recoverability
- 2PL does not guarantee recoverability
- This non-recoverable execution is 2-phase locked
  wl1[x] w1[x] r1[x] r2[x] c2
  - 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, T1 must commit before it’s first unlock-write (wu1): w1[x] c1 wu1[x] r2[x] c2
  - Stuck, can’t set rl3[y])

Automating Locking
- 2PL can be hidden from the application
- When a data manager gets a 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

2PL Preserves Transaction Handshakes
- Recall the definition: Ti commits before Tk starts
- 2PL serializes 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.
  - r1[x] w1[x] c1 r1[x] c1 w1[y] c1
  - T2 commits before T3 starts, but the only equivalent serial execution is T3 T1 T2
  - rl1[x] r1[x] w1[y] ru1[x] w1[x] c2
  - Stuck, can’t set rl3[y]) r1[y] ... so not 2PL
2PL Preserves Transaction Handshakes (cont’d)

- Stating this more formally …
- Theorem:
  For any 2PL execution $H$,
  there is an equivalent serial execution $H_s$,
  such that for all $T_i, T_k$,
  if $T_i$ committed before $T_k$ started in $H$,
  then $T_i$ precedes $T_k$ 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 $T_k$.

5.5 Implementing Two-Phase Locking

- Even if you never implement a DB system, it’s valuable to understand locking 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

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>$[T_1, r]$</td>
<td>$[T_1, w]$</td>
</tr>
<tr>
<td>y</td>
<td>$[T_4, w]$</td>
<td>$[T_3, w]$ $[T_6, r]$</td>
</tr>
</tbody>
</table>

Lock Manager (cont’d)

- Caller generates data-item-id, e.g. by hashing data item name
- The lock table is hashed on data-item-id
- Lock and Unlock must be atomic, so access to the lock table must be “locked”
- Lock and Unlock are called frequently. They must be very fast. Average < 100 instructions.
  - This is hard, in part due to slow compare-and-swap operations needed for atomic access to lock 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)
Deadlocks

- A set of transactions is deadlocked if every transaction in the set is blocked and will remain blocked unless the system intervenes.
  - Example: \( r_1[x] \) granted, \( r_2[y] \) granted, \( w_2[x] \) blocked, \( w_1[y] \) blocked and deadlocked.
- Deadlock is 2PL’s way to avoid non-SR executions.
  - \( r_1[x] r_1[y] r_2[y] r_2[y] \) ... can’t run \( w_2[x] w_1[y] \) 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.
- Often advocated in operating systems.
- Useless for TP, because it would require running transactions serially.
  - Example: to prevent the previous deadlock, \( r_1[x] r_1[y] w_1[x] w_1[y] \), the system can’t grant \( r_2[y] \).
- Avoiding 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.

Detection

- Detection approach: Detect deadlocks automatically, and abort a deadlocked transactions (the victim).
- It’s the preferred approach, because it
  - allows higher resource utilization and
  - uses cheaper algorithms.
- Timeout-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 = \( \{T_i \rightarrow T_j \mid T_i \text{ is waiting for } T_j \text{ to release a lock} \}\)
  - Example (previous deadlock): \( T_1 \sim T_2 \)
  - Theorem: If there’s a deadlock, then the waits-for graph has a cycle.

Detection Using Waits-For Graph (cont’d)

- So, to find deadlocks
  - when a transaction blocks, add an edge to the graph
  - periodically check for cycles in the waits-for graph.
- Don’t test for deadlocks too often. (A cycle won’t disappear until you detect it and break it.)
- When a deadlock is detected, select a victim from the cycle and abort it.
- Select a victim that hasn’t done much work (e.g., has set the fewest locks).

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.
  - \( T_1 \) restarts, bumps into \( T_2 \) and again deadlocks.
  - \( T_2 \) (the oldest) is aborted ...
- Choosing the youngest in a cycle as victim avoids cyclic restart, since the oldest transaction is never the victim.
- Can combine with other heuristics, e.g. fewest-locks.
MS SQL Server

- Aborts the transaction that is “cheapest” to roll back.
  - “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 many data managers
- 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 Locking

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

```
Node 1
| rl[x] | wl1[x] (blocked) |

Node 2
| rl[y] | wl2[y] (blocked) |
```

- Timeout-based detection is popular. Its weaknesses are less important in the distributed case:
  - aborts unnecessarily and some deadlocks persist too long
  - possibly abort 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 Dist’d Deadlock Detection

- Use waits-for graph cycle detection with a central deadlock detection server
  - more work than timeout-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 \(\Rightarrow\) SG cycle (modulo spontaneous aborts)
- Path pushing - Send paths \(T_i \rightarrow \ldots \rightarrow T_k\) to each node where \(T_k\) might be blocked.
  - Detects short cycles quickly
  - Hard to know where to send paths.
  - Possibly too many messages

Locking Granularity

- **Granularity** - size of data items to lock
  - e.g., files, pages, records, fields
- Coarse granularity implies
  - 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 implies
  - 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)

- 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

MGL Type and Instance Graphs

Database
  - Area
    - File
      - Record
        - R1.1
        - R1.2
        - R2.1
        - R2.2
        - R2.3
        - R2.4
  - A1
  - A2

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

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>y</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>y</td>
</tr>
<tr>
<td>riw</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

- 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 nth 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.

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. Hot Spot Techniques
8. Query-Update Techniques
9. Phantoms
10. B-Trees
11. Tree locking
5.6 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., Ti = read(x) … 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

Throughput

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# of Active Txns

Thrashing

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 \( 1/\dot{t} \) txns/second.

Start \( \rightarrow \) Lock \( L \) \( \rightarrow \) Commit \( \rightarrow \) Set the lock later in the transaction’s execution (e.g., defer updates till commit time)

To increase throughput, reduce \( t \) (lock holding 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

\[ \begin{array}{c|c|c|c|c} 
\text{Part#} & \text{Price} & \text{OnHand} & \text{PartName} & \text{CatalogPage} \\
\hline 
\end{array} \]

\[ \begin{array}{c|c|c|c|c} 
\text{Part#} & \text{Price} & \text{OnHand} & \text{Part#} & \text{PartName} & \text{CatalogPage} \\
\hline 
\end{array} \]

Mathematical Model of Locking

- \( K \) locks per transaction
- \( D \) lockable data items
- \( T \) time between lock requests
- \( N \) 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 \( K^2N/2D \).
- Probability of a deadlock is proportional to \( K^4N/D^2 \)
  - \( \text{Prob}(\text{deadlock})/\text{Prob}(\text{conflict}) = K^2/D \)
  - if \( K=10 \) and \( D = 10^6 \), then \( K^2/D = .0001 \)

5.7 Hot Spot Techniques

- If each txn holds a lock for \( t \) seconds, then the max throughput is \( 1/\dot{t} \) 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>r</th>
<th>w</th>
<th>inc</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<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.

\[
\text{bEnough} = \text{Verify}(\text{iQuantity}, n); \\
\text{If (bEnough) Decrement(iQuantity, n)} \\
\text{else print ("not enough"));}
\]

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)

5.8 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., rl1[x] r1[x] ru1[x] w1[x] w2[x] wu2[x] rl1[x] r1[x] ru1[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

MS SQL Server

- 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 ≤ T-Oldest, its versions are committed
  - For every aborted transaction with tid ≤ T-Oldest, its versions are wiped out
- Queries don’t need to know tids ≤ 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 ≥ 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” query 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 T1 and T2 modify each other’s readset concurrently?

Oracle Concurrency Control (cont’d)

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

5.9 Phantoms

- Problems when using 2PL with inserts and deletes

<table>
<thead>
<tr>
<th>Accounts</th>
<th>Assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acct#</td>
<td>Location</td>
</tr>
<tr>
<td>1</td>
<td>Seattle</td>
</tr>
<tr>
<td>2</td>
<td>Tacoma</td>
</tr>
<tr>
<td>3</td>
<td>Tacoma</td>
</tr>
</tbody>
</table>

T1: Read Accounts 1, 2, and 3
T2: Insert Accounts 4, Tacoma, 100
T3: Read Assets(Tacoma), returns 500
T4: Write Assets(Tacoma), returns 600
T5: Read Assets(Tacoma), returns 600
T6: Commit

The phantom record
The Phantom Phantom Problem
• It looks like T1 should lock record 4, which isn’t there!
• Which of T1’s operations determined that there were only 3 records?
  – Read end-of-file?
  – Read record counter?
  – SQL Select operation?
• This operation conflicts with T2’s Insert Accounts[4,Tacoma,100]
• Therefore, Insert Accounts[4,Tacoma,100] shouldn’t run until after T1 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.

5.10 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 → B, from field values to block numbers.
  – V = social security numbers. B = {1 .. 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
• K₁ < K₂ < … < Kᵢ₋₁ < Kᵢ
• Pᵢ points to a node containing keys < Kᵢ
• Pᵢ points to a node containing keys in range [Kᵢ₋₁, Kᵢ)
• So, Kᵢ₋₁ < Kᵢ₋₂ < … < Kᵢ₋₂ < Kᵢ₋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 (409³)
  – 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

<table>
<thead>
<tr>
<th>19</th>
<th>12</th>
<th>14</th>
<th>17</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

| 127 | 496 |
| 221 | 352 |
| 221 | 245 | 320 |

**5.11 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

A
B
C
D
E
F
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
  - P
  - 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)

<table>
<thead>
<tr>
<th>19</th>
<th>12</th>
<th>14</th>
<th>17</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

- \( r_1[P] \ r_2[P] \ r_2[C] \ w_2[C] \ w_2[C'] \ w_2[P] \ r_1[C] \ r_1[C'] \)

<table>
<thead>
<tr>
<th>19</th>
<th>12</th>
<th>14</th>
<th>17</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>