Concurrency Control

Concurrency Control

Yanlei Diao
UMass Amherst
March 27 and 29, 2007

Outline

- Serializability & recoverability
- Lock management
- Specialized locking techniques
  - Locking for the phantom problem
  - Efficient B+tree locking
- Levels of Isolation
- Concurrency control without locking
  - Optimistic CC
  - Timestamp-based CC
  - Multiversion CC

Conflict Serializability

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule
Example

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

- The schedule is not conflict serializable:

```
T1 ---- A ---- T2
   |       |
   |       |
   B
```

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Dependency Graph

- **Dependency graph:**
  - One node per Xact;
  - Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj’s actions (RW, WR, WW).

- **Theorem:** Schedule is conflict serializable if and only if its dependency graph is acyclic

Review: Strict 2PL

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each Xact must obtain a S (shared) lock on object before reading, an X (exclusive) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

- **Strict 2PL allows only schedules whose precedence graph is acyclic**
  - Therefore, Strict 2PL only allows serializable schedules!
Nonstrict 2PL

- Two-Phase Locking Protocol
  - Each Xact must obtain a S (shared) lock on object before reading, an X (exclusive) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Nonstrict 2PL ensures acyclicity of precedence graph is acyclic
  - Nonstrict 2PL only allows serializable schedules.
  - An equivalent serial schedule is given by the order of xacts entering their shrinking phase.

Weaker Condition on Serializability

- Conflict serializability is sufficient but not necessary for serializability.

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2
View Serializability (Contd.)

- A schedule is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is view serializable.
  - The converse is not true.
- Every view serializable schedule that is not conflict serializable contains a blind write.

Recoverability

- A schedule $S$ is recoverable if each xact commits only after all xacts from which it read have committed.

  | T1: R(A),W(A)       | Abort |
  | T2: R(A),W(A)       | Commit|
  | **Unrecoverable**   |       |

  | T1: R(A),W(A)       | Abort |
  | T2: R(A),W(A)       | Abort |
  | **Recoverable, but with cascading aborts** |       |

Recoverability (Contd.)

- $S$ avoids cascading rollback if each xact may read only those values written by committed xacts.

  | T1: R(A),W(A)       | Abort |
  | T2: R(A)            | W(A)  | Commit |
  | **Recoverable, no cascading aborts, but update of A by T2 is always lost** |       |
Recoverability (Contd.)

- S is **strict** if each xact may *read and write* only objects previously written by committed xacts.
  - No cascading aborts.
  - Actions of aborted xacts can be simply undone by restoring the original values of modified objects.
- Strict 2PL is recoverable, in addition to conflict serializability.

Venn Diagram for Schedules

Outline

- Serializability & recoverability
- Lock management
- Specialized locking techniques
  - Locking for the phantom problem
  - Efficient B+tree locking
- Levels of isolation
- Concurrency control without locking
  - Optimistic CC
  - Timestamp-based CC
  - Multiversion CC
Lock Management

- **Lock manager** handles the lock and unlock requests
- **Transaction table**: `xact id → <locks held in xact>`
- **Lock table**: `object id → lock table entry`
  - Object can be a page, a record, etc.
  - Lock table entry:
    - Number of transactions currently holding a lock
    - Type of lock held (shared or exclusive)
    - Queue of lock requests

Lock and Unlock

- When an xact requests a lock on an object O
  - If an S lock is requested, the queue of requests is empty, O is not currently locked in the X mode, then…
  - If an X lock is requested, no xact currently holds a lock on O, then…
  - Otherwise, …
- When an xact aborts or commits,
  - it releases all its locks and
  - when the lock on an object O’ is released, its lock table entry is updated and the request at the head of the queue is answered.

Other Lock Management Issues

- Lock and unlock have to be **atomic** operations.
  - Important when several instances of the lock manager code can execute concurrently.
  - Implemented using an OS synchronization mechanism such as a semaphore.
- **Lock upgrade**: An xact that holds a shared lock can be upgraded to hold an exclusive lock.
  - Important for avoiding deadlocks.
- **Convoys**: An xact that holds a heavily used lock can be suspended by the OS. Every other xact requesting the lock is queued and the queue can grow very long.
  - Drawback of building a DBMS on top of a general-purpose OS with preemptive scheduling.
**Deadlocks**

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

**Deadlock Prevention**

- Assign priorities based on timestamps.
  - The lower the timestamp, the higher the xact’s priority.
- Assume Ti wants a lock that Tj holds. Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts. *Lower priority xacts can never wait.*
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits. *Higher priority xacts never wait.*
- If a transaction re-starts, make sure it has its original timestamp.

**Deadlock Detection**

- Create a *waits-for graph*:
  - Nodes are transactions.
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock.
  - Note the difference from the dependency graph for conflict serializability.
- Periodically check for cycles, indicating deadlocks, in the waits-for graph.
  - Resolve a deadlock by aborting a transaction on the cycle and releasing all its locks.
**Deadlock Detection (Contd.)**

T1: S(A), R(A), S(B)  
T2: X(B), W(B)  
T3: S(C), R(C), X(A)  
T4: X(B)

**Multiple-Granularity Locks**

- Granularity of locks: concurrency vs. overhead
  - Finer granularity increases concurrency
  - It, however, increases overhead in set/clear operations and storage of the lock(s)
- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn’t have to decide! Because data “containers” are nested:

**Solution: New Lock Modes, Protocol**

- Allow Xacts to lock at each level, but with a special protocol using new “intention” locks:
  - Before locking an item, Xact must set “intention locks” on all its ancestors. Why?
  - SIX mode: Like S & IX at the same time.
  - To lock, request top-down.
  - To unlock, release at EOT or go from specific to general (i.e., bottom-up).
Multiple Granularity Lock Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
  - What if Xact holds SIX on parent? S on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

The protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.

Outline

- Serializability & recoverability
- Lock management
- Specialized locking techniques
  - Locking for the phantom problem
  - Efficient B+tree locking
- Levels of Isolation
- Concurrency control without locking
  - Optimistic CC
  - Timestamp-based CC
  - Multiversion CC
The Phantom Problem

- If we consider inserts and updates of records in a DB, even Strict 2PL won’t assure serializability:
  - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor, say, age = 71.
  - T2 next inserts a new sailor: rating = 1, age = 96.
  - T1 now reads the oldest sailor again; now, age = 96 appears as a phantom!

- Problem: T1 assumes that it has locked the set of all sailor records with rating = 1.
  - Assumption only holds if no sailor records are added or have the rating updated while T1 is executing!

The Phantom Problem (Contd.)

- The phantom problem can occur even if all xacts follow strict 2PL, often through index lookup.
  - Example with a newly inserted record?
  - Example with the update of an existing record?
- Need a mechanism to enforce the assumption that T1 indeed holds locks on all tuples/pages satisfying a condition.
  - Index locking, and
  - Predicate locking, a more general solution.

Index Locking

- If there is a dense index on the rating field using Alternative (2), T1 should lock the index page containing the data entries with rating = 1.
  - If there are no records with rating = 1, T1 must lock the index page where such a data entry would be, if it existed!
- If there is no suitable index, T1 must lock the entire file/table to prevent new pages from being added.
  - so that no new records with rating = 1 can be added.
Predicate Locking

- Fundamental reason for the phantom problem:
  - The old transaction model consists of reads and writes to individual data items.
  - In practice, transactions include queries that dynamically define sets of items based on predicates.
  - When the query is executing, all the records satisfying this predicate at a particular time can be locked.
  - Locking individual items, however, cannot prevent later addition of a record satisfying this predicate.
- Solution: extend lockable objects to index pages and further to arbitrary predicates!

Predicate Locking

- Grant locks on some logical predicate
  - e.g. age > 2*sal, or (age > 50 or age <30) and sal >10K
- Index locking is a special case of predicate locking
  - An existing index matches the predicate.
  - It supports efficient implementation of the predicate lock.
- In general, predicate locking has a lot of overhead.
  - For each record, we need to check if it satisfies a complex predicate.
  - Therefore, it is not commonly used.

Locking in B+ Trees

- How can we efficiently lock a particular leaf node?
  - BTW, don’t confuse this with multiple granularity locking!
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.
**Two Useful Observations**

- Higher levels of the tree only directly search for leaf pages.
- For inserts, a node on a path from root to leaf must be locked in X mode, only if a split can propagate up to it from the modified leaf.
  - Similar point holds w.r.t. deletes.
- Exploit these observations to design efficient locking protocols that guarantee serializability even though they violate 2PL.

**A Simple Tree Locking Algorithm**

- **Search**: Start at root and go down; repeatedly, lock child then unlock parent.
  - Searches never go back up. “Crabbing”, i.e. holding at most two locks on the parent and current nodes, is enough.
- **Insert/Delete**: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
  - If child is safe, release all locks on ancestors.
  - O.w., hold X locks up to the closed safe ancestor or the root.
- **Safe node**: Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.

**Example**

![Example Diagram](image)
A Better Tree Locking Algorithm
(See Bayer-Schkolnick paper)

- **Search**: As before.
- **Insert/Delete:**
  - Set locks as if for Search, get to leaf, and set X lock on leaf.
  - If leaf is not safe, release all locks, and restart Xact using previous Insert/Delete protocol.
- Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful. In practice, better than previous alg.

---

Example

Outline

- Serializability & recoverability
- Lock management
- Specialized locking techniques
  - Locking for the phantom problem
  - Efficient B+tree locking
- Levels of Isolation
- Concurrency control without locking
  - Optimistic CC
  - Timestamp-based CC
  - Multiversion CC
Levels of Isolation of Transactions*

Definition 1 using the notion of dirty data
- A write is **committed** when an xact is finished; o.w. the write is **dirty**.
- Xact T sees **level 0 of isolation** if:
  - T does not overwrite dirty data of other xacts.
- Xact T sees **level 1 of isolation** if it sees level 0 and
  - T does not commit any writes before EOT.
- Xact T sees **level 2 of isolation** if it sees level 1 and
  - T does not read dirty data of other xacts.
- Xact T sees **level 3 of isolation** if it sees level 2 and
  - Other xacts do not dirty any data read by T before T completes.

Called “degrees of consistency” in the paper written by Gray et al.

Levels of Isolation (Contd.)

Definition 2 using lock protocols
- Xact T sees **level 0 of lock protocol** if:
  - T sets a (possibly short) X lock on any data it dirties.
- Xact T sees **level 1 of lock protocol** if:
  - T sets a long X lock on any data it dirties.
- Xact T sees **level 2 of lock protocol** if:
  - T sets a long X lock on any data it dirties.
  - T sets a (possibly short) S lock on any data it reads.
- Xact T sees **level 3 of lock protocol** if:
  - T sets a long X lock on any data it dirties.
  - T sets a long S lock on any data it reads.

Levels of Isolation (Contd.)

Definition 2' using well-formedness and two phase
- T is **well-formed** w.r.t. to writes (reads) if it always locks an item in X (S) mode before writing (reading) it.
- T is **two phase** w.r.t. to writes (reads) if it does not X (S) lock after unlocking any item.
- Xact T sees **level 0 of lock protocol** if:
  - T is well-formed w.r.t. writes.
- Xact T sees **level 1 of lock protocol** if:
  - T is well-formed and two phase w.r.t. writes.
- Xact T sees **level 2 of lock protocol** if:
  - T is well-formed w.r.t. reads and writes.
  - T is two phase w.r.t. writes.
- Xact T sees **level 3 of lock protocol** if:
  - T is well-formed and two phase w.r.t. reads and writes.
Assertions

- Assertion 1:
  - If an xact observes the lock protocol definition of isolation (Def. 2), it is assured of the definition of isolation based on committed and dirty data (Def. 1).
  - Unless an xact actually sets the locks described by level 1 (2 or 3) of isolation, one can construct xact mixes and schedules that will cause the xact to run at a lower level of isolation.

- Assertion 2:
  - Each xact can choose its level of isolation as long as all xacts observe at least level 0 protocols.

Transaction Support in SQL-92

- Each transaction has an access mode (Read Only or not), and an isolation level.

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Anomaly</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
<td></td>
</tr>
<tr>
<td>2. Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
<td></td>
</tr>
<tr>
<td>3. Repeatable Reads</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
<td></td>
</tr>
<tr>
<td>4. Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

```
SET TRANSACTION ISOLATION LEVEL Serializable Read Only
```

Outline

- Serializability & recoverability
- Lock management
- Specialized locking techniques
  - Locking for the phantom problem
  - Efficient B+tree locking
- Levels of Isolation
- Concurrency control without locking
  - Optimistic CC
  - Timestamp-based CC
  - Multiversion CC
Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Phantom problem.
  - Lock contention for heavily used objects causes blocking.
- Idea: If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts commit.

Kung-Robinson Model

- Xacts have three phases:
  - READ: Xacts read from the database, but make changes to local (private) copies of objects.
  - VALIDATE: Check for conflicts. If there is a conflict, abort (clear the local copy & restart)!
  - WRITE: Make local copies of changes public.

Validation

- Goal: guarantee that only serializable schedules result.
- Technique: actually find an equivalent serializable schedule
  - assign an Xact id (timestamp) to each xact at the beginning of the validation phase, and
  - check if the timestamp-ordering of xacts is an equivalent serial order.
- ReadSet(Ti): Set of objects read by Xact Ti.
- WriteSet(Ti): Set of objects modified by Ti.
- Three test conditions sufficient to ensure an equivalent serializable schedule.
Test 1

- For all \( i \) and \( j \) such that \( T_i < T_j \), check that \( T_i \) completes before \( T_j \) begins.

\begin{align*}
T_i & \quad T_j \\
R & \quad V & \quad W \\
R & \quad V & \quad W
\end{align*}

Test 2

- For all \( i \) and \( j \) such that \( T_i < T_j \), check that:
  - \( T_i \) completes before \( T_j \) begins its Write phase
  - \( \text{WriteSet}(T_i) \cup \text{ReadSet}(T_j) \) is empty.

\begin{align*}
T_i & \quad T_j \\
R & \quad V & \quad W \\
R & \quad V & \quad W
\end{align*}

Does \( T_j \) read dirty data? Does \( T_i \) overwrite \( T_j \)'s writes?

- Is it correct?
  Each condition has to guarantee that the three classes of conflicts (W-R, R-W, W-W) go one way only.

Test 3

- For all \( i \) and \( j \) such that \( T_i < T_j \), check that:
  - \( T_i \) completes Read phase before \( T_j \) does
  - \( \text{WriteSet}(T_i) \cup \text{ReadSet}(T_j) \) is empty
  - \( \text{WriteSet}(T_i) \cup \text{WriteSet}(T_j) \) is empty.

\begin{align*}
T_i & \quad T_j \\
R & \quad V & \quad W \\
R & \quad V & \quad W
\end{align*}

Does \( T_j \) read dirty data? Does \( T_i \) overwrite \( T_j \)'s writes?

- Is it correct?
Comments on Optimistic CC

- Optimistic CC vs. Locking (pessimistic)
  - Optimistic CC: assume no conflicts first and only fix things when conflicts appear, by restarting xacts.
  - Locking: xacts are prevented in advance, by blocking, from (potentially) nonserializable actions.

- Works well for certain workloads:
  - All xacts are readers.
  - The interference among xacts is low. E.g. large amount of data, each xact accessing a small (likely non-overlapping) amount of data.

- Deadlock free, but may have starvation.
- No phantom problem!

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation
  - Code for validation is in a critical section, and critical section can reduce concurrency.
- Must make validated writes ‘global’.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.
  - Starvation may occur.

Timestamp CC

- Idea: (1) give each Xact a timestamp $TS$ when it begins; (2) give each object a read (write) timestamp $RTS$ ($WTS$) from most recent xact that reads (writes) it.
  - If action $a_i$ of Xact $Ti$ conflicts with action $a_j$ of Xact $Tj$, and $TS(Ti) < TS(Tj)$, then $a_i$ must occur before $a_j$.
  - Otherwise, restart violating Xact.
When Xact T wants to “Read” Object O

- If TS(T) < WTS(O), this violates timestamp order between T and most recent writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)
  - Contrast use of timestamps for deadlock prevention.

- If TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to \( \max(\text{RTS}(O), \text{TS}(T)) \)
- Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

When Xact T wants to “Write” Object O

- If TS(T) < RTS(O), this violates timestamp order of T w.r.t. writer of O.
  - Abort and restart T.

- If TS(T) < WTS(O) (TS(T) >= RTS(O)), violates timestamp order of T w.r.t. writer of O.
  - Thomas Write Rule: Can safely ignore such outdated writes; need not restart T! (T’s write is effectively followed by another write, with no intervening reads.)
  - Allows some serializable but non-conflict serializable schedules:

- Else, allow T to write O

Timestamp CC and Recoverability

- Unfortunately, unrecoverable schedules are allowed:
- Timestamp CC is modified to allow only recoverable schedules:
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.
- Similar to writers holding X locks until commit, but still not quite 2PL.
- Mostly used in distributed DB systems.
Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between xacts can be detected in the dependency graph.
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages).
- Naïve locking strategies may have the phantom problem.

Summary (Contd.)

- Index locking is common, and affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).
- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.
  - Bayer-Schkolnick illustrates potential for improvement.
- SQL-92 provides different isolation levels that control the degree of concurrency.

Summary (Contd.)

- Optimistic CC aims to minimize CC overheads in an "optimistic" environment where reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.
- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block Xacts, similar to locking.