Concurrent Control

R &G - Chapter 19

Smile, it is the key that fits the lock of everybody's heart.

- Anthony J. D'Angelo,
  The College Blue Book

Review

- DBMSs support concurrency, crash recovery with:
  - ACID Transactions
  - Log of operations
- A serial execution of transactions is safe but slow
  - Try to find schedules equivalent to serial execution
- One solution for serializable schedules is 2PL
Conflict Serializable Schedules

- **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

- **A schedule that is not conflict serializable:**

<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>

  ![Dependency graph]

  - 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 operation of Ti conflicts with an operation of Tj and Ti’s operation appears earlier in the schedule than the conflicting operation of Tj.

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

### Review: Strict 2PL

**Lock Compatibility Matrix**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **Strict Two-phase Locking (Strict 2PL) Protocol**:
  - Each Xact must obtain a S (shared) lock on object before reading, and 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**
Two-Phase Locking (2PL)

- **Two-Phase Locking Protocol**
  - Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - A transaction cannot 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.

Lock Management

- **Lock and unlock requests are handled by the lock manager**
- **Lock table entry:**
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- **Locking and unlocking have to be atomic operations**
- **Lock upgrade:** transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems
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. 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
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits
- **If a transaction re-starts, make sure it gets its original timestamp**
  - Why?
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
- **Periodically check for cycles in the waits-for graph**

Deadlock Detection (Continued)

Example:

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
<th>T3:</th>
<th>T4:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A), S(D), S(B)</td>
<td>X(B)</td>
<td>S(D), S(C), X(C)</td>
<td>X(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing transactions T1, T2, T3, and T4 with waiting and holding relationships]
Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to make same decision for all transactions!
- Data “containers” are nested:

```
    Database
     | contains
     | Tables
     |     | Pages
     |     |     | Tuples
```

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.
  - For unlock, go from specific to general (i.e., bottom-up).
  - SIX mode: Like S & IX at the same time.
  - Why useful?

```
<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IX</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIX</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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.

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 get X lock on tuples that are updated.
  - 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.
  - OR, T3 could behave like T2; can use lock escalation to decide which.
Locking in B+ Trees

• What about locking indexes --- why is it needed?
• Tree-based indexes present a potential concurrency bottleneck:
• If you ignore the tree structure & just lock pages while traversing the tree, following 2PL.
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.
• How can we efficiently lock a particular leaf node?
  - Btw, don't confuse this with multiple granularity locking!

Two Useful Observations

• 1) In a B+Tree, higher levels of the tree only direct searches for leaf pages.
• 2) For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)

• We can exploit these observations to design efficient locking protocols that guarantee serializability even though they violate 2PL.
A Simple Tree Locking Algorithm: “crabbing”

- **Search**: Start at root and go down; repeatedly, lock child then unlock parent.
- **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.
- **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**

```
Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*
```
A Better Tree Locking Algorithm  
(From 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.**

Dynamic Databases – The “Phantom” Problem

- **If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:**
- **Consider T1 - “Find oldest sailor for each rating”**
  - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71).
  - Next, T2 inserts a new sailor; rating = 1, age = 96.
  - T2 also deletes oldest sailor with rating = 2 (and, say, age = 80), and commits.
  - T1 now locks all pages containing sailor records with rating = 2, and finds oldest (say, age = 63).
- **No serial execution where T1’s result could happen!**
The Problem

• **T1 implicitly assumes that it has locked the set of all sailor records with rating = 1.**
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)

• **Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!**

Predicate Locking

• **Grant lock on all records that satisfy some logical predicate, e.g. age > 2*salary.**
• **Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.**
  - What is the predicate in the sailor example?
• **In general, predicate locking has a lot of locking overhead.**
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 all pages, and lock the file/table to prevent new pages from being added, to ensure that no records with rating = 1 are added or deleted.

Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- Locking is “pessimistic” because it assumes that conflicts will happen.
- If conflicts are rare, we might get better performance 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 private copies of objects.
  - **VALIDATE:** Check for conflicts.
  - **WRITE:** Make local copies of changes public.

Validation

- **Test conditions that are sufficient to ensure that no conflict occurred.**
- **Each Xact is assigned a numeric id.**
  - Just use a *timestamp*.
- **Xact ids assigned at end of READ phase, just before validation begins.** (Why then?)
- **ReadSet(Ti):** Set of objects read by Xact Ti.
- **WriteSet(Ti):** Set of objects modified by Ti.
Test 1

• For all i and j such that $T_i < T_j$, check that $T_i$ completes before $T_j$ begins.

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
  + WriteSet($T_i$) $\cap$ ReadSet($T_j$) is empty.

Does $T_j$ read dirty data? Does $T_i$ overwrite $T_j$’s writes?
Test 3

• For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty
  - WriteSet(Ti) ∩ WriteSet(Tj) is empty.

Ti

```
R V W
```

Tj

```
R V W
```

Does Tj read dirty data? Does Ti overwrite Tj’s writes?

Applying Tests 1 & 2: Serial Validation

• To validate Xact T:

```java
valid = true;
// S = set of Xacts that committed after Begin(T)
// The following is done in critical section
foreach Ts in S do {
  if ReadSet(T) intersects WriteSet(Ts)
    then valid = false;
}
if valid then {install updates; // Write phase
               Commit T }
else Restart T
```
Comments on Serial Validation

• Applies Test 2, with T playing the role of Tj and each Xact in T's (in turn) being Ti.
• Assignment of Xact id, validation, and the Write phase are inside a critical section!
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- can't happen.
  - If Write phase is long, major drawback.

• Optimization for Read-only Xacts:
  - Don't need critical section (because there is no Write phase).

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, and must make validated writes "global”.
  - Critical section can reduce concurrency.
  - 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.
**Timestamp CC**

- **Idea:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.

**Summary**

- **Correctness criterion for isolation is “serializability”**.
  - In practice, we use “conflict serializability”, which is somewhat more restrictive but easy to enforce.
- **There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Locks directly implement the notions of conflict.**
  - The lock manager keeps track of the locks issued.
    - Deadlocks can either be prevented or detected.
- **Must be careful if objects can be added to or removed from the database (“phantom problem”).**
- **Index locking common, affects performance significantly.**
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).
Summary (Contd.)

- **Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages);**
  - should not be confused with tree index locking!

- **Tree-structured indexes:**
  - Straightforward use of 2PL very inefficient.
  - Idea is to use 2PL on data to ensure serializability and use other protocols on tree to ensure structural integrity.
  - Bayer-Schkolnick illustrates potential for improvement.

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.**
- **There are many other approaches to CC that we don’t cover here. These include:**
  - timestamp-based approaches
  - multiple-version approaches
  - semantic approaches