Concurrency Control

CMPSCI 645
Apr 3, 2008

Slide content adapted from Ramakrishnan & Gehrke, Zack Ives
Review: the ACID Properties

- Particularly important: ensuring ACID properties
  - **Atomicity**: each operation looks atomic to the user
  - **Consistency**: each operation in isolation keeps the database in a consistent state (this is the responsibility of the user)
  - **Isolation**: should be able to understand what’s going on by considering each separate transaction independently
  - **Durability**: updates stay in the DBMS!!!
Review: properties of schedules

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.
  - Conflict-serializability
  - View-serializability
- **Recoverable schedule**: if Tj reads data written by Ti, then *Ti commits before Tj commits*.
- **Cascadeless schedule**: if Tj reads data written by Ti, then *Ti commits before read operation of Tj*.
Today

- Enforcing desirable schedules
  - Lock-based
    - Strict 2PL, 2PL
    - Phantoms
    - Index locking
  - Weak consistency in SQL
Lock-Based Concurrency Control

- DBMS must ensure
  - only serializable, recoverable schedules are allowed
  - No actions of committed trans lost while undoing aborted trans
- Lock - associated with some object
  - shared or exclusive
- Locking protocol - set of rules to be followed by each transaction to ensure good properties.
Lock Compatibility Matrix

Locks on a data item are granted based on a lock compatibility matrix:

<table>
<thead>
<tr>
<th>Request mode</th>
<th>Mode of Data Item</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Shared</td>
<td>Y</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Y</td>
</tr>
</tbody>
</table>

When a transaction requests a lock, it must wait (block) until the lock is granted.
## Transaction performing locking

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
</tr>
<tr>
<td><strong>R</strong>(A)</td>
</tr>
<tr>
<td>W(A)</td>
</tr>
<tr>
<td><strong>W</strong>(A)</td>
</tr>
<tr>
<td>unlock(A)</td>
</tr>
<tr>
<td>lock-S(B)</td>
</tr>
<tr>
<td><strong>R</strong>(B)</td>
</tr>
<tr>
<td>unlock(B)</td>
</tr>
</tbody>
</table>
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 can not request additional locks once it releases any locks.
    - growing phase
    - shrinking phase
Strict Two Phase Locking (Strict 2PL)

- **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.
  - A transaction can not request additional locks once it releases any locks.
    - growing phase
    - shrinking phase
  - All X (exclusive) locks acquired by a transaction must be held until commit.
Not admissible under 2PL

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Lock-based protocols

- **2PL ensures conflict serializability**
  - Transactions can be ordered by their end of growing phase (called *lock point*)
  - A 2PL schedule is equivalent to the serial schedule where transactions ordered by lock point order.
- **Strict 2PL ensures conflict serializable and cascadeless schedules**
  - Writers hold an X lock until they commit.
### Schedule following strict 2PL

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td></td>
<td>S(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(C)</td>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(B) Commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(C) Commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(C) Commit</td>
<td></td>
</tr>
</tbody>
</table>
Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry (for an object):
  - 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
Deadlocks

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A)</td>
<td>X(B)</td>
</tr>
<tr>
<td>X(B)</td>
<td>X(A)</td>
</tr>
</tbody>
</table>

- Deadlock must be **prevented** or **avoided**.
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
  - Add edge when queueing a lock request,
  - Remove edge when granting lock request.

- Periodically check for cycles in the waits-for graph
Deadlock Detection (Continued)

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

Diagram:

- T1 -> T2
- T2 -> T3
- T4 -> T2
- T3 -> T4

- T1
- T2
- T3
- T4
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 has its original timestamp.
Performance of Locking

- Lock-based schemes resolve conflicting schedules by **blocking** and **aborting**
  - in practice few deadlocks and relatively few aborts
  - most of penalty from blocking
- To increase throughput
  - lock smallest objects possible
  - reduce time locks are held
  - reduce hotspots
What should we lock?

T1

SELECT S.rating, MIN(S.age) 
FROM Sailors S 
WHERE S.rating = 8 

T2

UPDATE 
Sailors(Name, Rating, Age) 
VALUES ("Joe", 8, 33) 

- T1 S-lock on Sailors; T2 X-lock on Sailors
- T1 S-lock on all rows with rating=8; T2 X-lock on Joe’s tuple.
Phantom

- T1: “Find oldest sailor for each of the rating levels 1 and 2”
  - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71).
- T2: “Insert new sailor. rating=1, age=96”
- T2: “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).
The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with \textit{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!

- Strict 2PL will not assure serializability
The Phantom Problem

- Phantom problem: A transaction retrieves a collection of tuples and sees different results, even though it did not modify the tuples itself.
  - Conceptually: must lock all possible rows.
  - Can lock entire table.
  - Better, use index locking.
Index Locking

- If there is an 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 new records with rating = 1 are added.
Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. $age > 2 \times salary$.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
- In general, predicate locking has a lot of locking overhead.
Locking in B+ Trees

- How can we efficiently lock a particular leaf node?
- 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 direct searches for leaf pages.
- 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

- **Search:** Start at root and go down; repeatedly, S 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) Insert 45*
Transaction support in SQL

- Transaction automatically started for SELECT, UPDATE, CREATE
- Transaction ends with COMMIT or ROLLBACK (abort)
- SQL 99 supports SAVEPOINTs which are simple nested transactions
Specify isolation level

- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is more expensive than “no isolation”
    - We can’t do as many things concurrently (or we have to undo them frequently)
  - For performance, we generally want to specify the most relaxed isolation level that’s acceptable
    - Note that we’re “slightly” violating a correctness constraint to get performance!
Specifying isolation level in SQL

SET TRANSACTION [READ WRITE | READ ONLY] ISOLATION LEVEL [LEVEL];

LEVEL = SERIALIZABLE
      REPEATABLE READ
      READ COMMITTED
      READ UNCOMMITTED

Less isolation

The default isolation level is SERIALIZABLE

Locks sets of objects, avoids phantoms
REPEATABLE READ

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.
- Phantoms possible: inserts of qualifying tuples not avoided.

Locks only individual objects
READ COMMITTED

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.
- Value read by T may be modified while T in progress.
- Phantoms possible.

X locks on written objects, held to end
S locks on read objects, but released immediately.
READ UNCOMMITTED

- Greatest exposure to other transactions
- Dirty reads possible
- Can't make changes: must be READ ONLY
- Does not obtain shared locks before reading
  - Thus no locks ever requested.
Acceptable Dirty Read

If we are just checking availability of an airline seat, a dirty read might be fine! (*Why is that?*)

Reservation transaction:

```sql
EXEC SQL select occupied into :occ
    from Flights
    where Num= '123' and date=11-03-99
    and seat='23f';
if (!occ) {EXEC SQL
    update Flights
    set occupied=true
    where Num= '123' and date=11-03-99
    and seat='23f';}
else {notify user that seat is unavailable}
```
Real systems

- IBM DB2, Informix, Microsoft SQL Server, Sybase all use Strict PL (or variants)
- Oracle use multi-version CC (we didn’t cover this).
- All deal with deadlocks using waits-for graph.
Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph.
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Naïve locking strategies may have the phantom problem.
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.

In practice, better techniques now known; do record-level, rather than page-level locking.