Database Systems

Concurrency Control

Feifei Li
University of Utah
Transactions

- Concurrent execution of user programs is essential.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.

- A user’s program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

- A *transaction* is the DBMS’s abstract view of a user program: a sequence of reads and writes.
Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

- Issues: Effect of interleaving transactions, and crashes.
Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.

- An important property: *atomic*. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
ACID Property of Xact

- Atomic
- Consistency
  - Run by itself must leave the DB in a consistent state (no IC violations)
- Isolation
  - “protected” from the effects of concurrently scheduled other transactions
- Durability
  - If Xact has successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.
Example, A banking database

- Consider two transactions (Xacts):

<table>
<thead>
<tr>
<th>T1:</th>
<th>BEGIN</th>
<th>A=A+100,</th>
<th>B=B-100</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>BEGIN</td>
<td>A=1.06*A,</td>
<td>B=1.06*B</td>
<td>END</td>
</tr>
</tbody>
</table>

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect must be equivalent to these two transactions running serially in some order.
Example (Contd.)

- Consider a possible interleaving (schedule):

  T1: \( A=A+100, \quad B=B-100 \)
  T2: \( A=1.06*A, \quad B=1.06*B \)

But what about:

  T1: \( A=A+100, \quad B=B-100 \)
  T2: \( A=1.06*A, \quad B=1.06*B \)

The DBMS’s view of the second schedule:

  T1: \( R(A), W(A), \quad R(B), W(B) \)
  T2: \( R(A), W(A), R(B), W(B) \)
Scheduling Transactions

- **Serial schedule**: Schedule that does not interleave the actions of different transactions.

- **Equivalent schedules**: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

  T1: R(A), W(A), R(B), W(B), Abort
  T2: R(A), W(A), C

- Unrepeatable Reads (RW Conflicts):

  T1: R(A), R(A), W(A), C
  T2: R(A), W(A), C
Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>W(A),</td>
<td>W(B), C</td>
</tr>
<tr>
<td>T2:</td>
<td>W(A), W(B), C</td>
<td></td>
</tr>
</tbody>
</table>
Lock-Based Concurrency Control

- **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
    - **(Non-strict) 2PL Variant:** Release locks anytime, but cannot acquire locks after releasing any lock.
  - 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 serializable schedules.
  - Additionally, it simplifies transaction aborts
  - **(Non-strict) 2PL** also allows only serializable schedules, but involves more complex abort processing
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.
- All locks held by a transaction are released when the transaction completes (commit or abort).
- 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/dependency graph is acyclic.

<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>
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.
  - If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

- Can result in Cascading Aborts!
  - STRICT (!!) 2PL “Avoids Cascading Aborts” (ACA)
Strict 2PL vs Non-Strict 2PL

![Diagram comparing Strict 2PL and Non-Strict 2PL locking mechanisms over time.](image)
Aborting a Transaction

- If a transaction $Ti$ is aborted, all its actions have to be undone. Not only that, if $Tj$ reads an object last written by $Ti$, $Tj$ must be aborted as well!
- Most systems avoid such **cascading aborts** by releasing a transaction’s locks only at commit time.
  - If $Ti$ writes an object, $Tj$ can read this only after $Ti$ commits.
- In order to **undo** the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.
- This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.
The Log

- The following actions are recorded in the log:
  - *Ti writes an object:* the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts:* a log record indicating this action.

- Log records are chained together by Xact id, so it’s easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.
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>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A),</td>
<td>R(B), W(B)</td>
</tr>
<tr>
<td>R(A), W(A), R(B), W(B)</td>
<td></td>
</tr>
</tbody>
</table>

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

Dependency graph
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.
We will NOT cover the rest in this course.
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 is “weaker” than conflict serializability!
- Every conflict serializable schedule is view serializable, but not vice versa!
- I.e. admits more legal schedules
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
  - Requires latches (“semaphores”), which ensure that the process is not interrupted while managing lock table entries
  - See an undergrad. course for implementations of semaphores
- 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:

T1:  S(A), S(D),   S(B)
T2:   X(B)    X(C)
T3:   S(D), S(C),   X(A)
T4:   X(B)
Deadlock Detection (cont.)

- In practice, most systems do detection
  - Experiments show that most waits-for cycles are length 2 or 3
  - Hence few transactions need to be aborted
  - Implementations can vary
    - Can construct the graph and periodically look for cycles
    - Can do a “time-out” scheme: if you’ve been waiting on a lock for a long time, assume you’re deadlock and abort
Things We’re Glossing Over

- What should we lock?
  - We assume tuples here, but that can be expensive!
  - If we do table locks, that’s too conservative
  - *Multi-granularity* locking

- Locking in indexes
  - don’t want to lock a B-tree root for a whole transaction!
  - actually do non-2PL “latches” in B-trees

- CC w/out locking
  - “optimistic” concurrency control
  - “timestamp” and multi-version concurrency control
  - locking usually better, though
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!
- Let’s try it and see!
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!
  - e.g. table locks
Grant lock on all records that satisfy some logical predicate, e.g. $\text{age} > 2\times \text{salary}$.

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.
- too expensive!
Instead of predicate locking

- Table scans lock entire tables
- Index lookups do “next-key” locking
  - physical stand-in for a logical range!
Deadlock Detection (cont.)

- In practice, most systems do detection
  - Experiments show that most waits-for cycles are length 2 or 3
  - Hence few transactions need to be aborted
  - Implementations can vary
    - Can construct the graph and periodically look for cycles
      - When is the graph created?
        - Either continuously or at cycle checking time
      - Which process checks for cycles?
        - Separate deadlock detector
    - Can do a “time-out” scheme: if you’ve been waiting on a lock for a long time, assume you’re deadlock and abort
Things We’re Glossing Over

- What should we lock?
  - We assume tuples here, but that can be expensive!
  - If we do table locks, that’s too conservative
  - *Multi-granularity* locking

- Mechanisms
  - Locks and Latches

- Repeatability
  - In a Xact, what if a query is run again?
  - Are more records (phantoms) tolerable?
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:
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new “intention” locks:
  - Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.

- **IS** – Intent to get S lock(s) at finer granularity.
- **IX** – Intent to get X lock(s) at finer granularity.
- **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 – 2 level hierarchy

- 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.
- Lock escalation
  - Dynamically asks for coarser-grained locks when too many low level locks acquired
Locks and Latches

- What’s common?
  - Both used to synchronize concurrent tasks

- What’s different?
  - Locks are used for *logical consistency*
  - Latches are used for *physical consistency*

- Why treat ‘em differently?
  - Database people like to *reason* about our data

- Where are latches used?
  - In a lock manager!
  - In a shared memory buffer manager
  - In a B+ Tree index
  - In a log/transaction/recovery manager
## Locks vs Latches

<table>
<thead>
<tr>
<th>Latches</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ownership</strong></td>
<td>Processes</td>
</tr>
<tr>
<td><strong>Duration</strong></td>
<td>Very short</td>
</tr>
<tr>
<td><strong>Deadlocks</strong></td>
<td>No detection - code carefully !</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>Cheap - 10s of instructions directly addressable</td>
</tr>
<tr>
<td><strong>Modes</strong></td>
<td>S, X</td>
</tr>
<tr>
<td><strong>Granularity</strong></td>
<td>Flat - no hierarchy</td>
</tr>
</tbody>
</table>