Concurrency Control

Why Have Concurrent Processes?
- Better transaction throughput, response time
- Done via better utilization of resources:
  - While one processes is doing a disk read, another can be using the CPU or reading another disk.
- DANGER DANGER! Concurrency could lead to incorrectness!
  - Must carefully manage concurrent data access.
  - There’s (much!) more here than the usual OS tricks!

Transactions
- Basic concurrency/recovery concept: a transaction (Xact).
  - A sequence of many actions which are considered to be one atomic unit of work.
- DBMS “actions”:
  - reads, writes
  - Special actions: commit, abort

The ACID Properties
- **Atomicity**: All actions in the Xact happen, or none happen.
- **Consistency**: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation**: Execution of one Xact is isolated from that of other Xacts.
- **Durability**: If a Xact commits, its effects persist.

Passing the ACID Test
- **Concurrency Control**:
  - Guarantees Consistency and Isolation, given Atomicity.
- **Logging and Recovery**:
  - Guarantees Atomicity and Durability.
- We’ll do C. C. first:
  - What problems could arise?
  - What is acceptable behavior?
  - How do we guarantee acceptable behavior?

Schedules
- **Schedule**: An interleaving of actions from a set of Xacts, where the actions of any 1 Xact are in the original order.
  - Represents some actual sequence of database actions.
  - Example: R¹(A), W₁(A), R₂(B), W₂(B), R₃(C), W₃(C)
  - In a complete schedule, each Xact ends in commit or abort.
- **Initial State + Schedule → Final State**
Acceptable Schedules

- One sensible “isolated, consistent” schedule:
  - Run Xacts one at a time, in a series.
  - This is called a serial schedule.
  - NOTE: Different serial schedules can have different final states; all are “OK” – DBMS makes no guarantees about the order in which concurrently submitted Xacts are executed.

Serializable schedules:
- Final state is what some serial schedule would have produced.
- Aborted Xacts are not part of schedule; ignore them for now (they are made to ‘disappear’ by using logging).

Serializable Violations

- Two actions conflict when 2 xacts access the same item:
  - W-R conflict: T2 reads something T1 wrote; T1 still active
  - R-W and W-W conflicts: Similar.
- WR conflict (dirty read):
  - Result is not equal to any serial execution!
  - T2 reads what T1 wrote, but it shouldn’t have!

More Conflicts

- RW Conflicts (Unrepeatable Read)
  - T2 overwrites what T1 read.

- WW Conflicts (Lost Update)
  - T2 overwrites what T1 wrote.

Now, Aborted Transactions

- Serializable schedule: Equivalent to a serial schedule of committed Xacts.
  - as if aborted Xacts never happened.
- Two Issues:
  - How does one undo the effects of an xact?
    • We’ll cover this in logging/recovery
  - What if another Xact sees these effects?!
    • Must undo that Xact as well!

Cascading Aborts

- Abort of T1 requires abort of T2!
  - Cascading Abort
- What about WW conflicts & aborts?
  - T2 overwrites a value that T1 writes.
  - T1 aborts: its “remembered” values are restored.
  - Lose T2’s write! We will see how to solve this, too.
- An ACA (avoids cascading abort) schedule is one in which cascading abort cannot arise.
  - A Xact only reads/writes data from committed Xacts.

Recoverable Schedules

- Abort of T1 requires abort of T2!
  - But T2 has already committed!
- A recoverable schedule is one in which this cannot happen.
  - i.e. a Xact commits only after all the Xacts it “depends on” (i.e. it reads from or overwrites) commit.
  - Recoverable implies ACA (but not vice-versa!).
- Real systems typically ensure that only recoverable schedules arise (through locking).
Locking: A Technique for C. C.

- Concurrency control usually done via locking.
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
  - This is a simplistic view; suffices for now.
  - Mode \( \in \{S,X\} \)
  - Lock compatibility table:
  - If a Xact can’t get a lock, it is suspended on a wait queue.

Two-Phase Locking (2PL)

- **2PL**:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - If T releases any lock, it can acquire no new locks!
- Locks are automatically obtained by DBMS.
- **Guarantees** serializability!
  - Why?

Strict 2PL

- **Strict 2PL**:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - Hold all locks until end of transaction.
- **Guarantees** serializability, and recoverable schedule, too!
  - Thus ensures ACA!

Precedence Graph

- A Precedence (or Serializability) graph:
  - Node for each committed Xact.
  - Arc from Ti to Tj if an action of Ti precedes and conflicts with an action of Tj.
- T1 transfers $100 from A to B, T2 adds 6%
  - \( R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B) \)

Conflict Serializability & Graphs

- **Theorem**: A schedule is conflict serializable if and only if its precedence graph is acyclic.
- **Theorem**: 2PL ensures that the precedence graph will be acyclic!
- Strict 2PL improves on this by avoiding cascading aborts, problems with undoing WW conflicts; i.e., ensuring recoverable schedules.
Lock Manager Implementation

- Question 1: What are we locking?
  - Tuples, pages, or tables?
  - Finer granularity increases concurrency, but also increases locking overhead.
- Question 2: How do you “lock” something??
- Lock Table: A hash table of Lock Entries.
  - Lock Entry:
    - OID
    - Mode
    - List: Xacts holding lock (or a count)
    - List: Wait Queue

Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - 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 consistent DB state where T1 is “correct”!

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, predicate locking, or table locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

Summary of Concurrency Control

- Concurrency control key to a DBMS.
  - More than just mutexes!
- Transactions and the ACID properties:
  - C & I are handled by concurrency control.
  - A & D coming soon with logging & recovery.
- Conflicts arise when two Xacts access the same object, and one of the Xacts is modifying it.
- Serial execution is our model of correctness.

Summary, cont.

- Serializability allows us to “simulate” serial execution with better performance.
- 2PL: A simple mechanism to get serializability.
  - Strict 2PL also gives us recoverability, ACA
- Lock manager module automates 2PL so that only the access methods worry about it.
  - Lock table is a big main-mem hash table
- Deadlocks are possible, and typically a deadlock detector is used to solve the problem.

Summary, cont.: SQL-92 support

<table>
<thead>
<tr>
<th>ISOLATION LEVEL</th>
<th>LOST UPDATE</th>
<th>DIRTY READ</th>
<th>UNREPEATABLE READ</th>
<th>PHANTOM</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted (0)</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>No S locks; writers need not run at higher levels</td>
</tr>
<tr>
<td>Read Committed (1)</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Strict 2PL; S locks, S locks released anytime</td>
</tr>
<tr>
<td>Repeatable Reads (2)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Strict 2PL on data</td>
</tr>
<tr>
<td>Serializable (3)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Strict 2PL on data and indexes (or predicate locking)</td>
</tr>
</tbody>
</table>
State of the Art (concurrency)

- CC in broadcast data environments
- Update propagation for replication
- CC in search trees (R trees, etc.)
- Distributed optimistic CC
- CC in real-time DBMS
- CC for “long” transactions
- Version management