Goals for Today

• Reason about the granularity of objects locked by transactions, the implications of new data on object locking

• Understand how optimistic concurrency control techniques decide if an interleaved schedule could have caused consistency issues

• Learn about transactions and locking in SimpleDB, and the role of the Buffer Manager

Isolation Levels in SQL Standard

• Idea: give users control over locking overhead incurred by their xacts
• Xacts can be specified with desired Isolation Level  
  — Also, access mode like “read-only” only gets S locks

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Does not get read locks, (not allowed to write objects)
Write locks held to commit. Get read locks, but release those right away
Strict 2PL. Locks before read & write, on individual objects
Strict 2PL. Gets locks before read/write, including on sets of objects (index locks)
Multiple-Granularity Locks: Motivation

- What is appropriate object granularity to lock (records vs. pages vs. tables?)

- **Examples**
  - Xact wants to lock entire table... lock each record? → a lot of locks to grab!
  - Xact wants to read only a few records... lock whole table? → limits concurrency!

  Idea: allow xacts to have *different* lock granularity, as needed

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 intention 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: Have S & IX at the same time on a node.

  Read tuples in relation, updating certain ones

Multiple Granularity Lock Protocol

- Each xact starts locking from the root of the hierarchy
  - To get S or IS lock on a node, must hold IS or IX on parent node
  - 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
Multiple Granularity Lock Protocol

- Example usage
  - Xact scans relation R, and updates a few tuples:
    - gets an IX on DB, SIX lock on R and IX on necessary pages, then get X lock on tuples that are updated
  - Xact uses index to read only part of relation R:
    - gets an IS lock on DB, R, and pages, and repeatedly gets an S lock on tuples of R (also lock hierarchy for index)

Using SIX in first example means second example isn’t blocked!

Lab 4: Lock-based Concurrency Control

- Goal of Lab 4: add page-level locking to SimpleDB
  - Strict 2PL
  - Shared and Exclusive locks
  - Dealing with deadlock
  - Dealing with BufferPool eviction (more next lecture)

Concurrency: How does it Happen?

- **Process**: executing instance of an program
  - Strick 2PL
  - Shared and Exclusive locks

- **Thread**: a path of execution within a process
  - Can be many threads within a process!
  - Threads have shared access to data structures within the process

Possibly concurrent:

<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>

T1 reads record 1 in Relation A
-- IS on DB, Relation, Page, then S on record 1

T2 modifies record 2 in Relation A
-- IX on DB, Relation, Page, then X on record 2

T3 reads all records in Relation A
-- IS on DB, S on Relation

Permissions.READ_ONLY vs READ_WRITE

Such as, say, a data structure managing Lock requests

Java: Thread Synchronization

• Thread synchronization in Java
  – Keyword synchronized

– Synchronize specific block of code:
  \[
  \text{synchronized(this) \{ } // \text{some code } \}
  \]

– Synchronize entire method:
  \[
  \text{private synchronized void flushPage(PageId pid) \{ } \\
  // \text{some code} \\
  \}
  \]

Skeleton code for Lock Manager and Buffer Pool already has these in place

Optimistic CC: Motivation

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 at commit

Lab 4: Skeleton Code

• Modified BufferPool.java
  – Lock Manager as private inner class
  – Example: BufferPool.getPage() will require that the transaction acquires a lock first!

• Lock table data structure, should be able to:
  – Given transactionId, which pages does it have locked?
  – Given a page Id, which xacts hold a lock on the page?
  – Given a page, which Permissions is it locked with?

OCC: 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 with other Xacts
  – WRITE: Make local copies of changes public

Confusing name! Reads and writes happen during this phase, but on private copy

Key idea: check in validate phase if the Xact has behaved in a serializable manner
Each Xact is assigned a numeric id
- Just use a **timestamp (call it \(T_i\))**
- Timestamps are **assigned at end of READ phase**, just before validation begins

- **Main question:** is the timestamp-ordering of xacts equivalent to a serial ordering

Check for conflicts regarding:
- \(\text{ReadSet}(T_i)\): Set of objects read by Xact \(T_i\)
- \(\text{WriteSet}(T_i)\): Set of objects modified by \(T_i\)

**What Can Go Wrong: Example 1**

- Serial order for T1 and T2 determined by order they start validate phase
  - Can think of each xact executing instantaneously when its validation starts
- It is possible that T2 read A before T1 wrote A... but we don’t know!
  - We have to **abort T2 just in case it didn’t see T1’s change**
    (violating the presumed serial order)

**What Can Go Wrong: Example 2**

- Serial order: T1, T2
  - Final value of A should be T2’s version
- If we let T2 validate, **T2 could write A before T1 does**
  - (violating the presumed serial order)
  - Must abort T2

**Validate Phase: Checking for Conflicts**

- In both examples, serial order should be: T1, T2
- Example 1 issue:
  - T1 was in write phase while T2 was reading, **and**
  - WriteSet(T1) overlaps ReadSet(T2)
- Example 2 issue:
  - T1 was in write phase while T2 tried validating, **and**
  - WriteSet(T1) overlaps WriteSet(T2)

**Need a set of tests to use to check when validation will be okay!**
Test 1 – Have non-overlapping xacts

• For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

Tj sees changes made by Ti, ok since Ti happened serially before it

Test 2 – Have no write phase overlap

• If Ti completes before Tj begins its Write phase, check:
  \( \text{WriteSet}(Ti) \cap \text{ReadSet}(Tj) \) is empty.

Test 3 – Have overlapping write phases

• If Ti completes Read phase before Tj does, check:
  \( \text{WriteSet}(Ti) \cap \text{ReadSet}(Tj) \) is empty
  AND \( \text{WriteSet}(Ti) \cap \text{WriteSet}(Tj) \) is empty.

Exercise 3

a) No, since Tj does not read anything Ti wrote
b) No, since Tj only read data that Ti didn’t write, Ti didn’t change a value that Tj read multiple times

c) No, since Ti finishes writing before Tj starts writing
Exercise 4 (a-c)

- Which transaction(s) will commit successfully in each schedule?

(a) Both will commit
(b) Both will commit
(c) T2 will abort, since its ReadSet overlaps T1’s WriteSet

Implementing OCC

- **Backwards serial validation** for a xact T_v:
  - Make sure serializability not violated with respect to all xacts T_i that committed after T_v started

- In practice, implement a xact’s validate and write phases together in a critical section
  - Only one xact can be executing its critical section at a time

Serial Validation: Applying Tests 1 & 2 (backwards validation)

- To validate Xact T_v:
  - valid = true;
  - // S = set of Xacts that committed after Start(T_v)
  - // (above definition implements Test 1)
  - // The following is done in critical section
  - < foreach T_s in S do {
    - if ReadSet(T_v) intersects WriteSet(T_s)
    - then valid = false;
    - }
  - if valid then { install updates; // Write phase
    - Commit T }
  - else Abort T

Reflections on Serial Validation

- Applies Test 2, with T_v playing the role of T_j and each Xact in T_s (in turn) being T_i
  - So T_v is looking “backwards” in time

- Assignment of Xact id (timestamp), Validation, and the Write phase are inside a critical section!
  - Nothing else goes on concurrently
  - So, no need to check for Test 3 --- cannot happen
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
  - Optimization for read-only xact: doesn’t need critical section
- Optimistic CC restarts Xacts that fail validation
  - Work done so far is wasted

But this might not happen that much!

Finding Order for Xacts

- Lock-based CC
  - Conflicting actions ordered by the order that xacts acquired locks
  - This effectively orders the xacts
- OCC (Kung-Robinson model)
  - Xacts ordered by time they enter validate phase
  - Validation checks that any conflicting actions occur in that same order
- Another approach: Timestamp CC
  - Idea: If action $a_i$ of Xact $T_i$ conflicts with action $a_j$ of Xact $T_j$, and $\text{Start}(T_j) < \text{Start}(T_i)$, then only okay if $a_i$ occurs before $a_j$

OK!

- Objects get a read-timestamp (RTS) and a write-timestamp (WTS)
- Each Xact gets timestamp (TS) when it starts

MVCC and Snapshot Isolation (SI)

- Multi-version concurrency control (MVCC)
  - Maintain several versions of objects with timestamps
  - Xact Ti reads most recent version of database before Ti started
- Snapshot Isolation is an implementation of MVCC
  - Xact Ti operates on its snapshot in isolation
  - Reading is never blocked! Reads don’t block writes
  - First Committer Wins policy for conflicting writes in xacts Ti and Tj

Some systems used to implement “Serializable” isolation level with SI