CS 133: Databases

Fall 2019
Lec 18 – 11/12
Transactions
Prof. Beth Trushkowsky

Goals for Today

• Discuss the “phantom problem” and options for Isolation levels in a DBMS

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

Warm-up Exercise

(See exercise sheet. You can start before class.)

Sequence 2:
T2 blocks on T1 on object A
T3 blocks on T2 on object B
T1 blocks on T2 on object B
DEADLOCK! Waits-for-graph has cycle between T1 and T2

“Dynamic” Databases

• Database is a static set of objects!

• With Insert and Delete possible, even Strict 2PL (on individual objects) will not assure serializability
The “Phantom” Problem – Example 1

• Consider T1 – “Find oldest sailor”
  T1 locks all Sailor records, finds oldest sailor (age = 71)

T2 inserts a new sailor; age = 96

T1 checks again for the oldest sailor, finds oldest sailor (age = 96)

T1 locks all Sailor records, finds oldest sailor (age = 71)

The sailor with age 96 is a “phantom tuple” from T1’s point of view --- first it’s not there, then it is

No serial execution where T1’s result could happen!

The “Phantom” Problem – Example 2

• Consider T3 – “Find oldest sailor for each rating”
  T3 locks all pages containing sailor records with rating = 1
  finds oldest sailor (age = 71)

T4 inserts a new sailor (new page); rating = 1, age = 96
T4 also deletes oldest sailor with rating = 2, age = 80

Commits

T3 now locks all pages containing sailor records with rating = 2, and finds oldest (age = 63).

• T3 saw only part of T4’s effects!

No serial execution where T3’s result could happen!

The Problem in “Phantom Problem”

• How do you lock something that does not yet exist??

• T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.
  – Assumption only holds if no sailor records are added while they are executing!
  – Need some mechanism to enforce this assumption, e.g., index locking (an implementation of predicate locking)

• Conflict serializability on reads and writes of individual objects guarantees serializability only if the set of objects is fixed

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, can sometimes access mode like “read-only” only gets S locks
Isolation Levels in SQL Standard

- SQL Standard defines levels based on what anomalies can be observed
- Implementation of levels varies!

<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>

Possible implementation

- 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)

Exercise 2

- T1: S(B)R(B)X(A)W, A, U(A)commit
- T2: S(A) R, A, U(A)

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
    - Key idea: check in validate phase if the Xact has behaved in a serializable manner... e.g., backwards validation
  - **WRITE**: Make local copies of changes public

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
OCC: Validate Phase

- Each Xact is assigned a numeric id
  - Just use a timestamp
  - Timestamps are assigned at end of READ phase, just before validation begins

- Main question: is the timestamp-ordering of xacts equivalent to some serial ordering?

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

  The DBMS also keeps track of timestamp when each Xact starts and finishes

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

- T2 read A in read phase, which could have happened before T1 wrote A (as shown above)
  - We have to abort T2 just in case it didn’t see T1’s change (violating the presumed serial order of T1,T2)

What Can Go Wrong: Example 2

- Presumed 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 test to use to check when validation will be okay!
Test 1 – Applicable when have *non-overlapping xacts*

- For all \(i\) and \(j\) such that \(T_i < T_j\), test passes if \(T_i\) completes before \(T_j\) begins.

![Diagram showing Ti and Tj]

\(T_j\) sees changes made by \(T_i\), ok since \(T_i\) happened serially before it

---

Test 2 – Applicable when Xacts overlap but have *no write phase overlap*

- If \(T_i\) completes before \(T_j\) begins its Write phase, passes if:
  \[\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)\] is empty.

![Diagram showing Ti and Tj]

---

**Exercise 3**

a) No, since \(T_j\) does not read anything \(T_i\) wrote

b) No, since \(T_j\) only read data that \(T_i\) *didn’t* write, \(T_i\) didn’t change a value that \(T_j\) read multiple times

c) No, since \(T_i\) finishes writing before \(T_j\) starts writing

---

Test 3 – Applicable when Xacts have *overlapping write phases*

- If \(T_i\) completes Read phase before \(T_j\) does, passes if:
  \[\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)\] is empty
  \[\text{AND} \text{WriteSet}(T_i) \cap \text{WriteSet}(T_j)\] is empty.

![Diagram showing Ti and Tj]

---

Does \(T_j\) read dirty data or have unrepeatable reads?
Does \(T_i\) overwrite \(T_j\)'s writes?
Exercise 4
start with (a)

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

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

To validate Xact \( T_v \):

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

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