CS 133: Databases

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

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

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

"Dynamic" Databases

- Database is a static set of objecs!
- 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
commits

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

No serial execution where T1's result could happen!

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

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!

Isolation Level	*Dirty Read*	*Unrepeatable Read*	*Phantom Problem*	***Possible implementation***
Read Uncommitted	Maybe	Maybe	Maybe	**Does not get read locks, (not allowed to write objects)**
Read Committed	No	Maybe	Maybe	**Write locks held to commit. Get read locks, but release those right away**
Repeatable Read	No	No	Maybe	**Strict 2PL. Locks before read & write, on individual objects**
Serializable	No	No	No	**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) S(A) R(A)U(A)Commit

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: 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! **Both** reads and writes happen, but on private copy of data

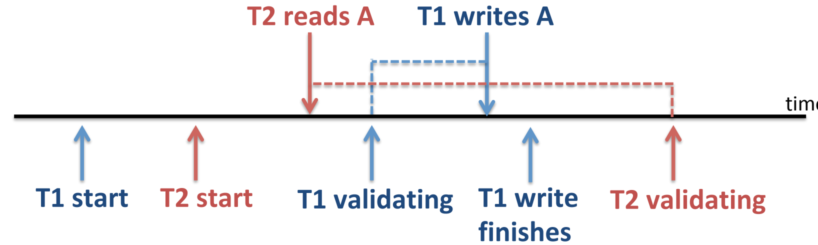
Key idea: check in validate phase if the Xact has behaved in a serializable manner… e.g., ***backwards validation***

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

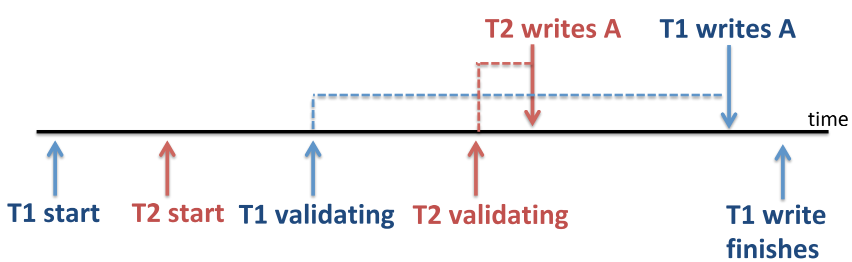
The DBMS also keeps track of timestamp when each Xact starts and finishes

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**

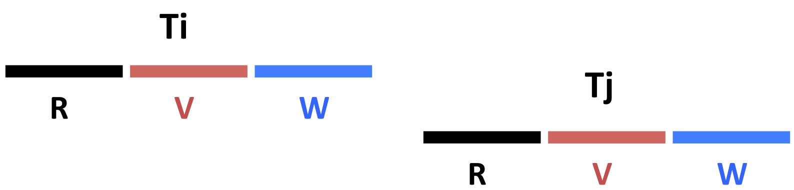
Validate Phase: Checking for Conflicts

- 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*

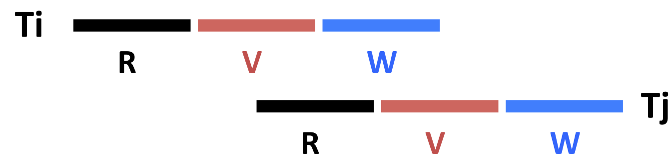
- For all i and j such that Ti < Tj, test passes if Ti completes before Tj begins.



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

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

- If Ti completes before Tj begins its Write phase, passes if:
  WriteSet(Ti) ∩ ReadSet(Tj) is empty.



Exercise 3

a) No, since Tj does not read anything Ti wrote

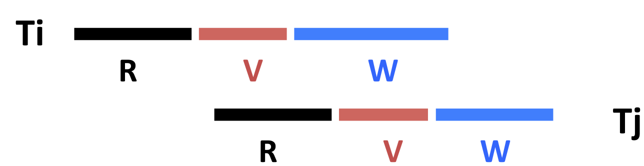
b) No, since Tj only read data that Ti *didn't* write, Ti didn't change a value that Tj read multiple times

c) No, since Ti finishes writing before Tj starts writing

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

Minimum criteria to consider Ti < Tj

- If Ti completes Read phase before Tj does, passes if:
  WriteSet(Ti) ∩ ReadSet(Tj) is empty
  AND WriteSet(Ti) ∩ WriteSet(Tj) is empty.



Does Tj read dirty data or have unrepeatable reads?
Does Ti overwrite Tj's writes?

Exercise 4
start with (a)

(a) Both will commit

(b) 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, 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**

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

Why not Test 3?

- 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
```

start of critical section

end of critical section