Lecture 22: Intro to Transactions & Logging IV

Professor Xiannong Meng
Spring 2018
Lecture and activity contents are based on what Prof Chris Ré of Stanford used in his CS 145 in the fall 2016 term with permission

2. Conflict Serializability, Locking & Deadlock

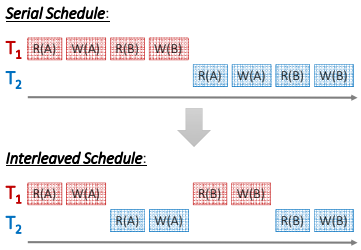
2

What you will learn about in this section

1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

3

Recall: Concurrency as Interleaving TXNs

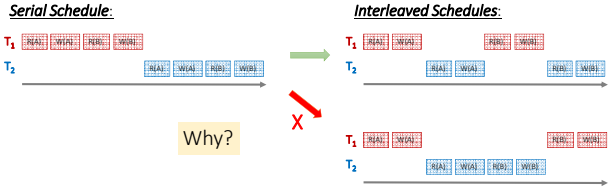


- For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

We call the particular order of interleaving a **schedule**

4

Recall: "Good" vs. "bad" schedules



Why?

We want to develop ways of discerning "good" vs. "bad" schedules

5

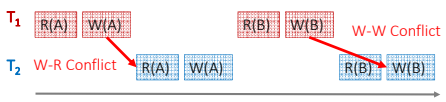
Ways of Defining "Good" vs. "Bad" Schedules

- Recall from last time: we call a schedule ***serializable*** if it is equivalent to *some* serial schedule
  - We used this as a notion of a "good" interleaved schedule, since **a serializable schedule will maintain isolation & consistency**
- Now, we'll define a stricter, but very useful variant:
  - ***Conflict serializability***

We'll need to define ***conflicts*** first..

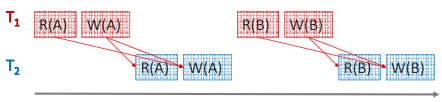
Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



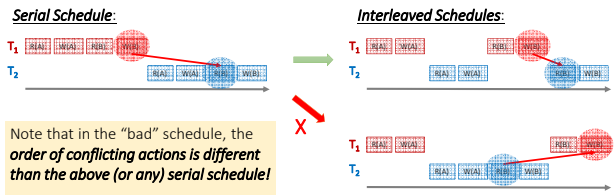
All "conflicts"!

Conflict Serializability

- Two schedules are **conflict equivalent** if:
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation!

Recall: "Good" vs. "bad" schedules



Note that in the "bad" schedule, the *order of conflicting actions is different than the above (or any) serial schedule!*

Conflict serializability also provides us with an operative notion of "good" vs. "bad" schedules!

10

Note: Conflicts vs. Anomalies

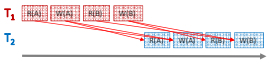
- **Conflicts** are things we talk about to help us characterize different schedules
  - Present in both "good" and "bad" schedules
- **Anomalies** are instances where isolation and/or consistency is broken because of a "bad" schedule
  - We often characterize different anomaly types by what types of conflicts predicated them

The Conflict Graph

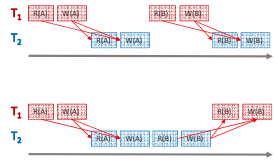
- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i precede and conflict with some actions in T_j**

What can we say about "good" vs. "bad" conflict graphs?

Serial Schedule:



Interleaved Schedules:



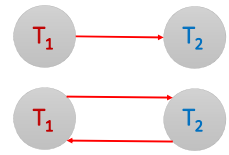
X

A bit complicated…

What can we say about "good" vs. "bad" conflict graphs?

Serial Schedule:

Interleaved Schedules:



X

Simple!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

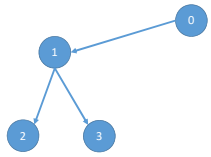
Let's unpack this notion of acyclic conflict graphs…

DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A directed **acyclic** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

DAGs & Topological Orderings

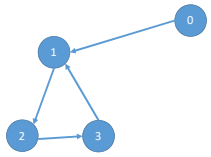
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



There is none!

Relation to conflict serializability

- In the conflict graph, a topological ordering of nodes corresponds to **a serial ordering of TXNs**
- Thus an **acyclic** conflict graph → conflict serializable!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

Strict Two-Phase Locking

- We consider **locking**- specifically, *strict two-phase locking*- as a way to deal with concurrency, because is **guarantees conflict serializability (if it completes- see upcoming…)**
- Also (*conceptually*) straightforward to implement, and transparent to the user!

Strict Two-phase Locking (Strict 2PL) Protocol:

TXNs obtain:

Two types of lock- "**exclusive**" and "**shared**" are used.

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds an X lock, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds an S lock , no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- "exclusive", "shared"- meant to be intuitive- no tricks!

Strict 2PL

Theorem: Strict 2PL allows only schedules whose dependency graph is acyclic

Proof Intuition: In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable ⇒ serializable schedules

Strict 2PL

- If a schedule follows strict 2PL and locking, it is conflict serializable…
  - …and thus serializable
  - …and thus maintains isolation & consistency!
- Not all serializable schedules are allowed by strict 2PL.
- So let's use strict 2PL, what could go wrong?

Two-phase Locking (2PL) Protocol:
A variation of Strict 2PL

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds an X lock, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds an S lock , no other TXN can get *an X lock* on that object
- TXNs cannot request any locks once it releases any locks

Note: 2PL says TXNs can release some locks before commit.

Deadlock Detection: Example

First, T_1 requests a shared lock on A to read from it

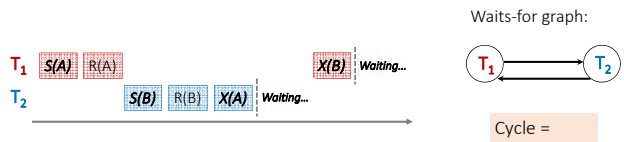
Deadlock Detection: Example

Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1...**

Deadlock Detection: Example



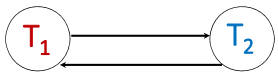
Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2... DEADLOCK!**

```
sqlite3.OperationalError: database is locked
```

```
ERROR:  deadlock detected
DETAIL:  Process 321 waits for ExclusiveLock on tuple of
relation 20 of database 12002; blocked by process 4924.
Process 404 waits for ShareLock on transaction 689; blocked
by process 552.
HINT:  See server log for query details.
```

The problem?
Deadlock!??!



Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  1. Deadlock prevention
  2. Deadlock detection

Deadlock Prevention

- There are a number of possible mechanisms to prevent deadlocks
  - Do not allow "hold-and-wait", the transaction holding a lock while requesting another must release the lock it is holding.
  - Set priorities among transactions so that the transaction with higher priority can take the lock away from the ones with lower priority.

Deadlock Detection

- Create the **waits-for graph**:
  - Nodes are transactions
  - There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*
- Periodically check for (***and break***) cycles in the waits-for graph

Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation** & **consistency** are maintained
  - We formalized a notion of **serializability** that captured such a "good" interleaving schedule
- We defined **conflict serializability**, which implies serializability
- **Locking** allows only conflict serializable schedules
  - If the schedule completes… (it may deadlock!)