Database Systems
CSE 414

Lecture 21: More Transactions
(Ch 8.1-3)

Announcements

- HW6 due on Today
- WQ7 (last!) due on Sunday
- HW7 will be posted tomorrow
  - due on Wed, May 24
  - using JDBC to execute SQL from Java
  - using SQL Server via Azure
  - setup covered in **section tomorrow**

Outline

- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Locks (18.3)

Review: Transactions

- Problem: An application must perform *several* writes and reads to the database, as a unit
- Solution: multiple actions of the application are bundled into one unit called a *Transaction*
- Turing awards to database researchers
  - Charles Bachman 1973 for CODASYL
  - Edgar Codd 1981 for relational databases
  - Jim Gray 1998 for transactions

Review: TXNs in SQL

```
BEGIN TRANSACTION
        [SQL statements]
COMMIT    or
ROLLBACK (=ABORT)
```

```
[single SQL statement]
```

If BEGIN… missing, then TXN consists of a single instruction

Review: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Isolation: The Problem

- Multiple transactions are running concurrently T_1, T_2, …
- They read/write some common elements A_1, A_2, …
- How can we prevent unwanted interference ?
- The SCHEDULER is responsible for that

Notation says nothing about tables…
(These techniques apply more generally.)

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Serial Schedule

- A *serial schedule* is one in which transactions are executed one after the other, in some sequential order
- Fact: nothing can go wrong if the system executes transactions serially
  - But database systems don't do that because we need better performance

Example

A and B are elements in the database
t and s are variables in txn source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

Time

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Another Serial Schedule

Time

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable Schedule

A schedule is *serializable* if it is equivalent to some serial schedule

A Serializable Schedule

T1	T2
READ(A, t) t := t+100 WRITE(A, t)	
	READ(A,s) s := s*2 WRITE(A,s)
READ(B, t) t := t+100 WRITE(B,t)	
	READ(B,s) s := s*2 WRITE(B,s)

This is a serializable schedule.
This is NOT a serial schedule

A Non-Serializable Schedule

T1	T2
READ(A, t) t := t+100 WRITE(A, t)	
	READ(A,s) s := s*2 WRITE(A,s) READ(B,s) s := s*2 WRITE(B,s)
READ(B, t) t := t+100 WRITE(B,t)	

How do We Know if a Schedule is Serializable?

Notation

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Key Idea: Focus on *conflicting* operations

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability

Conflicts: (it means: cannot be swapped)

Two actions by same transaction T_i:	$r_i(X)$; $w_i(Y)$
Two writes by T_i, T_j to same element	$w_i(X)$; $w_j(X)$
Read/write by T_i, T_j to same element	$w_i(X)$; $r_j(X)$ $r_i(X)$; $w_j(X)$

Conflict Serializability

- A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swaps of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable
- A serializable schedule may not necessarily be conflict-serializable

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); \boxed{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); \boxed{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); r_1(B); r_2(A); \boxed{w_2(A); w_1(B);} r_2(B); w_2(B)$

$\Downarrow$

….

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j

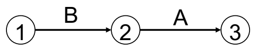
- The schedule is serializable iff the precedence graph is acyclic

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



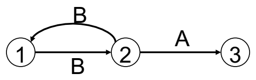
This schedule is conflict-serializable

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

Scheduler

- Scheduler = is the module that schedules the transaction's actions, ensuring serializability
- Also called Concurrency Control Manager
- We discuss next how a scheduler may be implemented

Implementing a Scheduler

Major differences between database vendors

- Locking Scheduler
  - Aka "pessimistic concurrency control"
  - SQLite, SQL Server, DB2, Spanner
- Multiversion Concurrency Control (MVCC)
  - Aka "optimistic concurrency control"
  - Postgres, Oracle, Spanner

We discuss only locking in 414

Locking Scheduler

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

By using locks scheduler ensures conflict-serializability

What Data Elements are Locked?

Major differences between vendors:

- Lock on the entire database
  - SQLite
- Lock on individual records
  - SQL Server, DB2, etc

Let's Study SQLite First

- SQLite is very simple
- More info: http://www.sqlite.org/atomiccommit.html

- Lock types
  - READ LOCK (to read)
  - RESERVED LOCK (to write)
  - PENDING LOCK (wants to commit)
  - EXCLUSIVE LOCK (to commit)

SQLite

Step 1: when a transaction begins

- Acquire a READ LOCK (aka "SHARED" lock)
- All these transactions may read happily
- They all read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock

---

SQLite

Step 2: when one transaction wants to write

- Acquire a RESERVED LOCK
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a RESERVED LOCK

SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can't coexists with *read locks*

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

Why not write to disk right now?

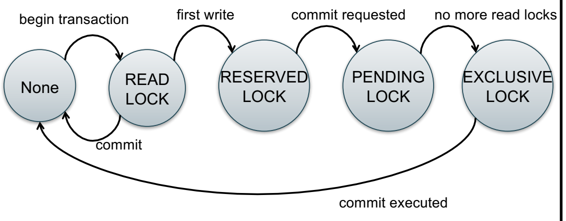
SQLite

Step 4: when all read locks have been released

- Acquire the EXCLUSIVE LOCK
- Nobody can touch the database now
- All updates are written permanently to the database file

- Release the lock and COMMIT

SQLite



SQLite Demo

create table R(a int, b int);
insert into R values (1,10);
insert into R values (2,20);
insert into R values (3,30);

Demonstrating Locking in SQLite

T1:
  begin transaction;
  select * from R;
  -- T1 has a READ LOCK
T2:
  begin transaction;
  select * from R;
  -- T2 has a READ LOCK

Demonstrating Locking in SQLite

T1:
  update R set b=11 where a=1;
  -- T1 has a RESERVED LOCK

T2:
  update R set b=21 where a=2;
  -- T2 asked for a RESERVED LOCK: DENIED

Demonstrating Locking in SQLite

T3:
  begin transaction;
  select * from R;
  commit;
  -- everything works fine, could obtain READ LOCK

Demonstrating Locking in SQLite

```
T1:
  commit;
  -- SQL error: database is locked
  -- T1 asked for PENDING LOCK -- GRANTED
  -- T1 asked for EXCLUSIVE LOCK -- DENIED
```

Demonstrating Locking in SQLite

```
T3':
  begin transaction;
  select * from R;
  -- T3 asked for READ LOCK-- DENIED (due to T1)

T2:
  commit;
  -- releases the last READ LOCK; T1 can commit
```