Introduction to Data Management
CSE 344

Lecture 19: More Transactions

Announcements

- HW7 (final one!) will be released today
  - Some Java programming required
  - Connecting to SQL Azure
  - Due ~~Wednesday~~, ~~Aug 7~~ Aug 10
    Thursday
- WQ7 (final one!) released
  - Due Monday, ~~Aug 10~~ Aug 7

Outline

- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Transaction implementation using 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*

Review: Transactions in SQL

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

```
[single SQL statement]
```

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

Know your ~~chemistry~~ transactions: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a DBMS state where integrity holds, to another where integrity holds
    - remember integrity constraints?
- 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

Rollback transactions

- If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK

- The DB returns to the state prior to the transaction

- Useful for atomicity – ROLLBACK on error.

Isolation: The Problem

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

Schedules

A schedule is a sequence of interleaved actions from all transactions

Review: Serial Schedule

- A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

Batch Mode

- Review: nothing can go wrong if the system executes transactions serially (up to what we have learned so far)
  - But DBMS don't do that because we want better overall system performance

Example

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

A must = B

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)

app code

A = 50
B = 50

Example of a (Serial) Schedule

Time

T1	T2	
READ(A, t)		
t := t+100		
WRITE(A, t)		A = 150
READ(B, t)		
t := t+100		B = 150
WRITE(B,t)		
	READ(A,s)	
	s := s*2	
	WRITE(A,s)	A = 300
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	B = 300

A = 50
B = 50

Another Serial Schedule

Time

T1	T2	
	READ(A,s)	
	s := s*2	
	WRITE(A,s)	A = 100
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	B = 100
READ(A, t)		
t := t+100		
WRITE(A, t)	A = 200	
READ(B, t)		
t := t+100		
WRITE(B,t)	B = 200	

Review: Serializable Schedule

A schedule is serializable if it is equivalent to a 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)

No conflicts

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

reads writes

Conflicts

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

Conflict Serializability

Conflicts: (i.e., swapping will change program behavior)

can not swap or re order

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

maintaine order of transactio i

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 swappings of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable

Conflict Serializability

Example:

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

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

Conflict Serializability

Example:

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

$$\Downarrow$$

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

Conflict Serializability

Example:

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

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

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

Conflict Serializability

Example:

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

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

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

….

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

Testing for Conflict-Serializability

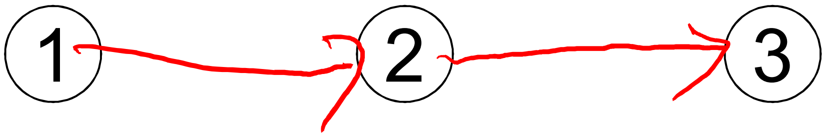
Precedence graph:

- 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

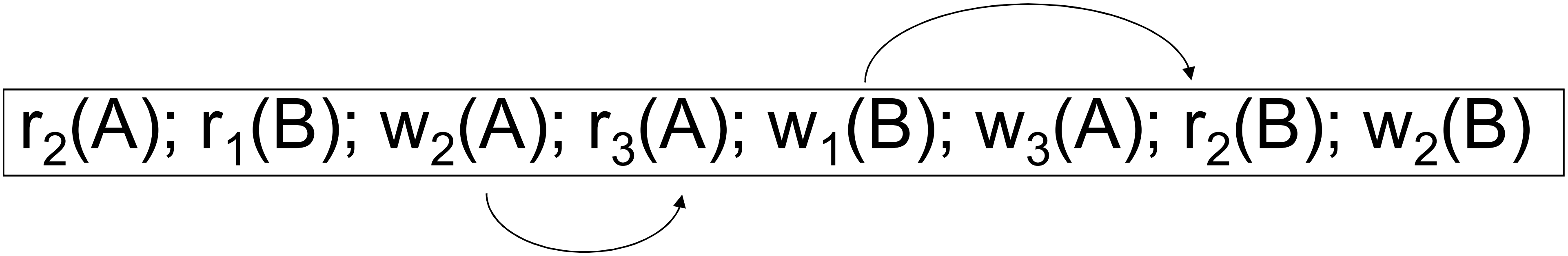
- The schedule is conflict-serializable iff the precedence graph is acyclic

Example 1

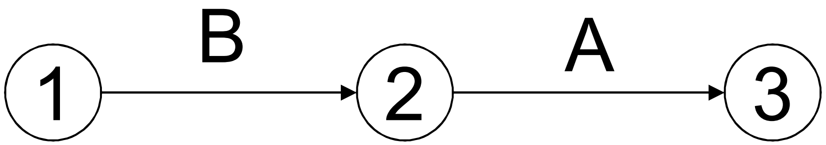
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



Example 1



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

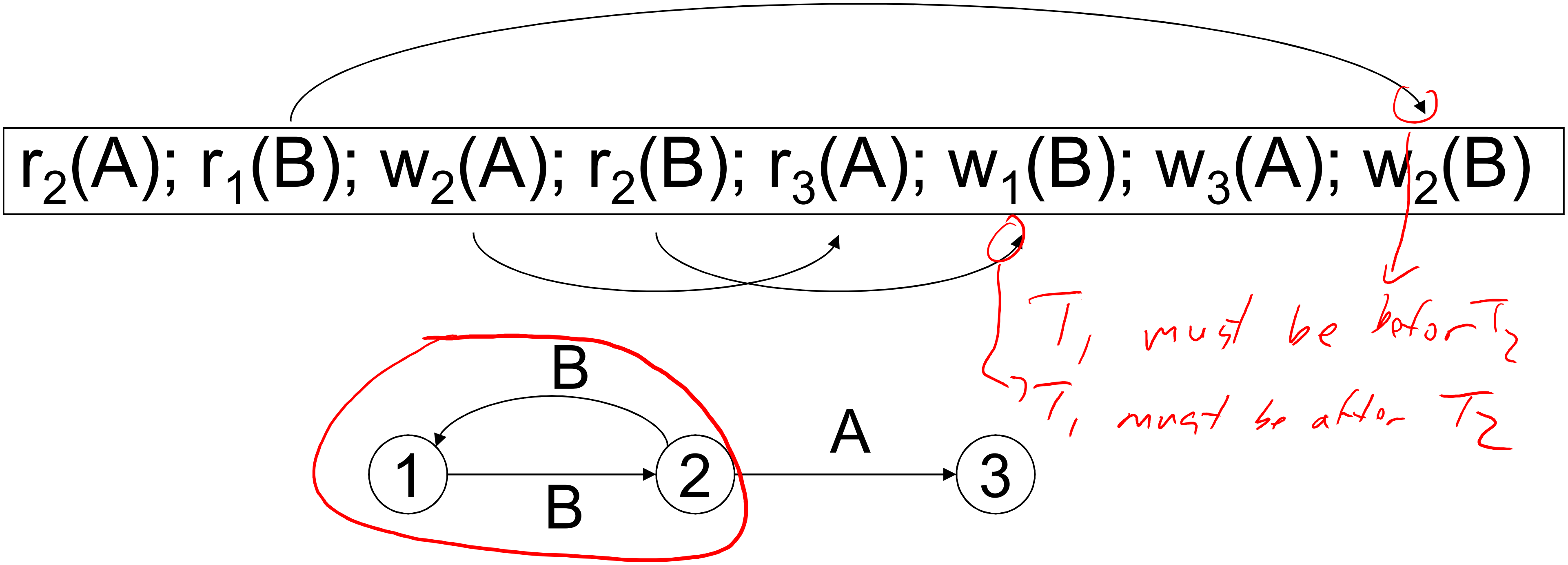


This schedule is conflict-serializable

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



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

T_1 must be before T_2

T_1 must be after T_2

This schedule is NOT conflict-serializable

Scheduler

- Scheduler = 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
- Multiversion Concurrency Control (MVCC)
  - Aka "optimistic concurrency control"
  - Postgres, Oracle

We discuss only locking schedulers in 344

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

Review: SQLite

- 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?

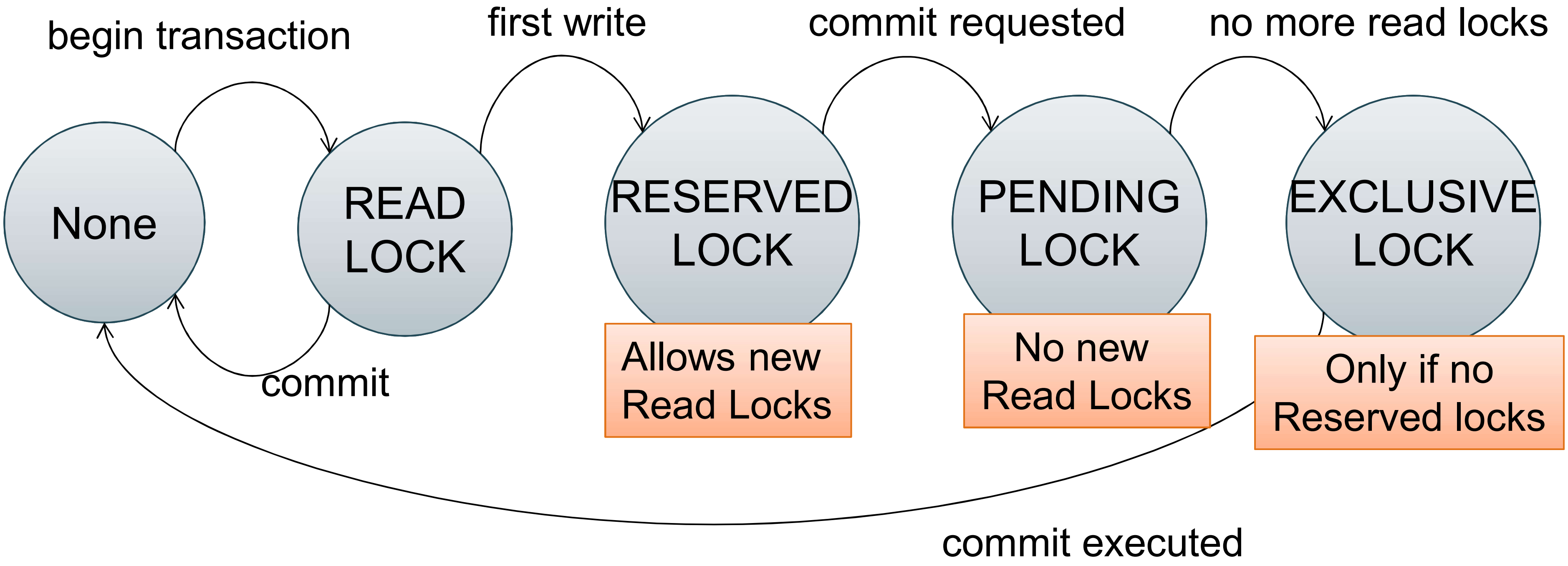
SQLite

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



Lecture notes contains a SQLite demo

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);
```

Does not seem to work in Ubuntu for Window
(looks like a file flush issue)

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

T_1 : Pending

T3':

begin transaction;

select * from r;

-- T3 asked for READ LOCK-- DENIED (due to T1)

T2:

commit;

Can not commit if any read locks

-- releases the last READ LOCK; T1 can commit

How do anomalies show up in schedules?

- What could go wrong if we didn’t have concurrency control:
  - Dirty reads (including inconsistent reads)
  - Unrepeatable reads
  - Lost updates

Many other things can go wrong too

Dirty Reads

Write-Read Conflict

T_1: WRITE(A)

T_2: READ(A)

T_1: ABORT

Inconsistent Read

Write-Read Conflict

T_1: A := 20; B := 20;	
T_1: WRITE(A)	
	T_2: READ(A);
	T_2: READ(B);
T_1: WRITE(B)	

Unrepeatable Read

Read-Write Conflict

T_2: READ(A);

T_1: WRITE(A)

T_2: READ(A);

Lost Update

Write-Write Conflict

T_1: READ(A)

T_1: A := A+5

T_1: WRITE(A)

T_2: READ(A);

T_2: A := A*1.3

T_2: WRITE(A);

Next time: handling anomalies with locks