Introduction to Data Management CSE 344

Lecture 19 and 20: Transactions

Announcements

- HW6 is due tomorrow
  - Try to finish on time so that you have more time for HW7 and HW8.
- HW7 is posted
  - You will learn “SQL in Java” for HW7 in the sections tomorrow, do not miss!
- WQ7 due next <u>Monday (note: NOT Tuesday)</u>
  - Last webquiz!
  - NOTE: You can see full explanations to all WQ questions after the due date.

Outline

- Basics (6.6, 1.2.4)
- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Locks (18.3) [Start today and finish next time]

Lecture 19:
(Up to slide 15)

MOTIVATION: SQLite in class

- See the notes
- Use SQL UPDATE to book flight seats for two passengers from two windows

- Schema:

Flights(seat, is_occupied)

Two customers could book the same seat.

What went wrong?

Challenge

- For performance, want to execute many applications concurrently.
- All these applications read and write data.
- But for correctness, multiple operations often need to be executed as an **atomic transaction** over the database.
- In our example, both users have reserved the same seat, and they are unhappy

What are the other problems?

- Write-Read Conflict
- Read-Write Conflict
- Write-Write Conflict
- System failure/crash

WRITE-READ Conflict

- Called "**Dirty read**" or "**Inconsistent Read**"
- One application is in the middle of performing some changes:
- **(a) A Manager is re-balancing budget and is moving money between projects:**
  - Step 1: Remove $10K from project 1
  - Step 2: Add $7K to project 2
  - Step 3: Add $3k to project 3
- **(b) The CEO wants to see the total balance,runs:**
  - **select sum(money) from Budget after Step 1**
  - The CEO sees "inconsistent" data

WRITE-READ Conflict

- A famous example of dirty reads:
  - Husband deposits $100 check but pretends like its $1M
  - System will detect the problem and will stop the deposit
  - BUT what if the wife withdraws $1M from ATM next door at the same time?
  - If this application manages to see the "dirty" $1M value... the bank is in trouble.

READ-WRITE Conflict

- **"Unrepeatable read“**
  - An application reads the value of some database item: e.g., inventory. Sees one book remaining, wants to buy.
  - Another application updates that value: e.g., someone else buys the last book and now the inventory is zero.
  - The first application re-reads the value and finds that it has changed... the inventory is now at zero.
  - This leads to "Unrepeatable read" or other anomalies

WRITE-WRITE Conflict

- "**Lost update**“
  - Account 1 = $100,   Account 2 = $100, Total = $200
  - Application 1 writes $200 to account 1 (without reading its balance).
  - Application 1 writes $0 to account 2
  - Application 2 writes $200 to account 2
  - Application 2 writes $0 to account 1
  - Final state (if executed one by one, in any order): one account has $200, the other one has $0
  - Total = $200 (unchanged)

WRITE-WRITE Conflict

- "**Lost update**“
- What if the applications executed concurrently:
  - Application 1 writes $200 to account 1
  - Application 2 writes $200 to account 2
  - Application 1 writes $0 to account 2
  - Application 2 writes $0 to account 1

  - Where did the money go?

System Failure/Crash

- That's not all…
- What if a failure happens while an application is updating the database? This can also create problems:
  - e.g., What if your browser crashes while you are purchasing a $1K gift for your pet?
  - What do you do?

Transaction Definition

- Transaction = a collection of statements that are executed atomically
- it looks like this:

```
begin transaction;
 . . .
 commit;  -- or rollback;
```

SQLite in classs

- Run the same UPDATE code as transactions
- Now Repeat the two transactions by user 1 and 2, but switch the commit order (try at home).
  - i.e., user 1 commits while user 2 continues the transaction. But user 1 receives an error when attempting to commit.
- Can you guess why we got an error?
  - (will learn soon)
- WARNING: You can see somewhat different behaviors with different DBMSs (more next lecture).

End of Lecture 19

Lecture 20

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 *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

ROLLBACK: Aborting Transactions

- If the app gets to a place where it can't complete the transaction successfully, it can execute ROLLBACK
- This causes the system to "abort" the transaction
- The database returns to the state without any of the previous changes made by activity of the transaction

- Can you guess some reasons for ROLLBACK?

ROLLBACK: Aborting Transactions

Reasons for Rollback

- User changes their mind ("ctl-C"/cancel)
- Explicit in program, when app program finds a problem
  - E.g. when the # of rented movies > max # allowed
- System-initiated abort
- System crash
- Housekeeping, e.g. due to timeouts

ACID Properties

A DBMS guarantees the following four properties of transactions:

- 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 (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID: Atomicity

- **Definition:** each transaction is ATOMIC meaning that all its updates must happen or not at all.
  - Important for recovery and if we need to abort a transaction in the middle.

ACID: Atomicity

- Example: move $100 from account 1 to account 2

```
update Accounts
set balance = balance – 100
where account = 1

update Accounts
set balance = balance  +100
where account = 2
```

- If the system crashes between the two updates, then we are in trouble.

ACID: Atomicity

begin transaction

update Accounts

set balance = balance – 100

where account = 1

update Accounts

set balance = balance +100

where account = 2

commit

- Now all updates happen atomically, when the commit is done.

ACID: Atomicity

More correct:

begin transaction

-- read the balance in account 1

-- if (balance < 100) **ROLLBACK** // Any update already performed is undone

-- else

-- update the two bank accounts

commit

ACID: Isolation

- A transaction executes concurrently with other transaction

- Isolation:

  the effect is as if each transaction executes in isolation of the others

- Will see in detail later today

ACID: Consistency

- Driven by application
  - e.g. for transferring money from one account to another, the sum should be the same

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - The system makes sure that the txn is atomic
  - When defining integrity constraints, it is possible to specify whether constraints can be delayed and checked only at the END of the transaction instead of being checked after each statement.

ACID: Durability

- The effect of a transaction must continue to exists after the transaction, or the whole program has terminated

- Means: write data to disk

Comments

- Think how ACID transactions help application development.
  - Will be helpful in hw7!

- By default, when using a DBMS, each statement is its own transaction!

Implementing ACID Properties

- Isolation:
  - Achieved by the *concurrency control* manager (or *scheduler*)
  - Discussed briefly in 344 today and in the next lecture
  - Discussed more extensively in 444
- Atomicity
  - Achieved using a *log* and a *recovery manager*
  - Discussed in 444
- Durability
  - Implicitly achieved by writing back to disk
- Consistency
  - Implicitly guaranteed by A and I

Last two properties implied by the first two

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

Schedules

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

Example

A and B are elements in the database
t and s are variables in tx 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)

Read value of A into s

“Serial” Schedule

- **Definition:** A SERIAL schedule of the transactions is one in which transactions are executed one after the other, in serial order
  - Fact: nothing can go wrong if the system executes transactions serially
  - But the database system doesn't do that for enabling better performace
    - Too many transactions, parallelism required
    - Assume the scenario when each customer of AA/United etc is forced to book tickets one by one!

A Serial Schedule

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)

Serializable Schedule

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

We want to ensure this instead of 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)	

Do you see why?

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)	

T2 should be executed after T1

T1 should be executed after T2

How do We Know if a Schedule is Serializable?

Notation

Write on B by transaction T1

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

Conflicts

Conflicts: pair of actions (in order) in schedule s.t. if swapped, then behavior changes.

Two actions by same transaction T_i: $\boxed{r_i(X); w_i(Y)}$

Two writes by T_i, T_j to same element $\boxed{w_i(X); w_j(X)}$

Read/write by T_i, T_j to same element $\boxed{w_i(X); r_j(X)}$ $\boxed{r_i(X); w_j(X)}$

Note: any #actions can appear between them

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

- Stronger condition than serializability
  - Every conflict-serializable schedule is serializable
  - A serializable schedule may not necessarily be conflict-serializable (see example on page 893 in the textbook)

Conflict Serializability

Example:

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

Are there any conflicts?

Conflict Serializability

Example:

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

NOTE: all conflicts are not shown.
Now we will check if it is conflict serializable

Conflict Serializability

Example:

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

$\Downarrow$

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

Conflict Serializability

Example:

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

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

Conflict Serializability

Example:

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

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

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

Conflict Serializability

Example:

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

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

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

….

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

Conflict Serializability

Next, how do we check for conflict serializability algorithmically?

Ans: Using Precedence Graph

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

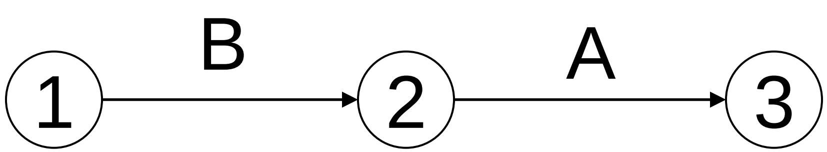
Example 1

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

1 2 3

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

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

1 2 3

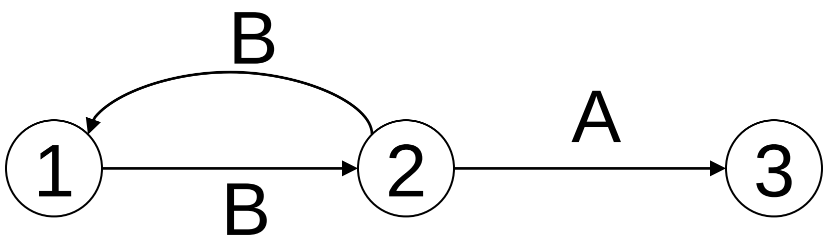
Example 2

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

1 2 3

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

We discuss only locking 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

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?

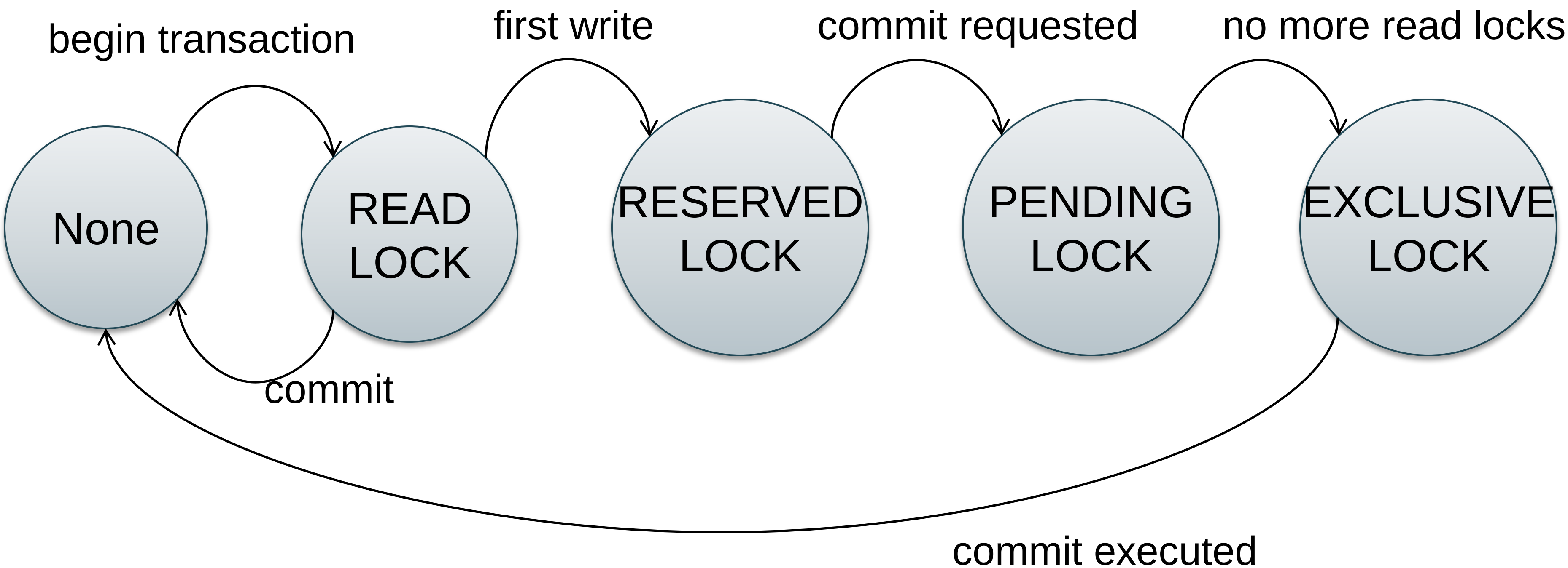
SQLite

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


begin transaction
first write
commit requested
no more read locks
None
READ LOCK
RESERVED LOCK
PENDING LOCK
EXCLUSIVE LOCK
commit
commit executed

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
```

T1: READ lock
T2: READ lock

T1: READ lock
T2: 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

T1: RESERVED lock
T2: READ lock

T1: RESERVED lock
T2: READ lock

Demonstrating Locking in SQLite

T3:

```
begin transaction;
select * from r;
-- everything works fine, could obtain READ LOCK
```

T1: RESERVED lock
T2: READ lock
T3: READ lock

T1: RESERVED lock
T2: READ lock
T3: READ lock

Demonstrating Locking in SQLite

T3:

commit;

T1: RESERVED lock
T2: READ lock

T1: RESERVED lock
T2: 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

T1: PENDING lock
T2: READ lock

T1: PENDING lock
T2: READ lock

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: PENDING lock

T1: PENDING lock

Demonstrating Locking in SQLite

T1:

commit;

-- T1 asked for EXCLUSIVE LOCK – GRANTED

-- No more locks on the database

T3‘:

select * from r;

T3’: READ lock

-- T3 asked for READ LOCK-- GRANTED

Try at home

- Is this schedule serializable ?
  – Note: only one element = whole database file

- If so, then what is the serialization order of the four transactions T1, T2, T3, T3' ?

Additional Review: Some Famous Anomalies

- 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

Additional Review: Dirty Reads

Write-Read Conflict

T_1: WRITE(A)

T_2: READ(A)

T_1: ABORT

Additional Review: 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)

Additional Review: Unrepeatable Read

Read-Write Conflict

T_2: READ(A);

T_1: WRITE(A)

T_2: READ(A);

Additional Review: 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);