CSE 444: Database Internals

Lectures 14
Transactions: Overview +
Concurrency Control using Locking

Outline

- Transactions motivation, definition, properties
  - 344 review
- Concurrency control and locking
  - Also 344 review

Motivating Example

```
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3
```

```
SELECT sum(money)
FROM Budget
```

Would like to treat each group of instructions as a unit

Definition

- **A transaction** = one or more operations, single real-world transition
- Examples
  - Transfer money between accounts
  - Purchase a group of products
  - Register for a class (either waitlist or allocated)

Transactions

- Major component of database systems
- Critical for most applications; arguably more so than SQL
- Turing awards to database researchers:
  - Charles Bachman 1973
  - Edgar Codd 1981 for inventing relational dbs
  - Jim Gray 1998 for inventing transactions

Transaction Example

```
START TRANSACTION
UPDATE Budget SET money = money - 100
WHERE pid = 1
UPDATE Budget SET money = money + 60
WHERE pid = 2
UPDATE Budget SET money = money + 40
WHERE pid = 3
COMMIT  (or ROLLBACK)
```

ROLLBACK

- 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
  - Database returns to a state without any of the changes made by the transaction

Reasons for Rollback

- User changes their mind ("ctl-C"/cancel)
- Explicit in program, when app program finds a problem
  - e.g. when qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - e.g. due to timeouts

ACID Properties

- Atomicity: Either all changes performed by transaction occur or none occurs
- Consistency: A transaction as a whole does not violate integrity constraints
- Isolation: Transactions appear to execute one after the other in sequence
- Durability: If a transaction commits, its changes will survive failures

What Could Go Wrong?

- Why is it hard to provide ACID properties?
- Concurrent operations
  - Isolation problems
  - We saw one example earlier
- Failures can occur at any time
  - Atomicity and durability problems
  - Later lectures
- Transaction may need to abort

Different Types of Problems

```
Client 1: INSERT INTO SmallProduct(name, price)
          SELECT pname, price
          FROM Product
          WHERE price <= 0.99

          DELETE Product
          WHERE price <=0.99

Client 2: SELECT count(*)
          FROM Product

          SELECT count(*)
          FROM SmallProduct
```

What could go wrong ? Inconsistent reads

Different Types of Problems

```
Client 1:
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = 'Gizmo'

Client 2:
        UPDATE Product
        SET Price = Price*0.5
        WHERE pname='Gizmo'
```

What could go wrong ? Lost update

Different Types of Problems

Client 1: UPDATE SET Account.amount = 1000000000
WHERE Account.number = 'my-account'

Aborted by system

Client 2: SELECT Account.amount
FROM Account
WHERE Account.number = 'my-account'

What could go wrong ? Dirty reads

Types of Problems: Summary

- Concurrent execution problems
  - Write-read conflict: dirty read (includes inconsistent read)
    - A transaction reads a value written by another transaction that has not yet committed
  - Read-write conflict: unrepeatable read
    - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
  - Write-write conflict: lost update
    - Two transactions update the value of the same object. The second one to write the value overwrite the first change
- Failure problems
  - DBMS can crash in the middle of a series of updates
  - Can leave the database in an inconsistent state

Outline

- Transactions motivation, definition, properties
- Concurrency control and locking

Schedules

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

Example

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

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

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)

Notice:
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)	

Ignoring Details

- Sometimes transactions' actions can commute accidentally because of specific updates
  - Serializability is undecidable !
- Scheduler should not look at transaction details
- Assume worst case updates
  - Only care about reads r(A) and writes w(A)
  - Not the actual values involved

Notation

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

Conflict Serializability

Conflicts:

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 swappings of adjacent non-conflicting actions

Example:

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

⇩

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

The Precedence Graph Test

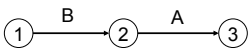
Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions T_i
- Edge from T_i to T_j if T_i makes an action that conflicts with one of T_j and comes first
- The test: if the graph has no cycles, then it is conflict serializable !

Example 1

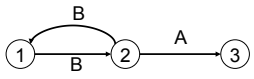
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

Example 2

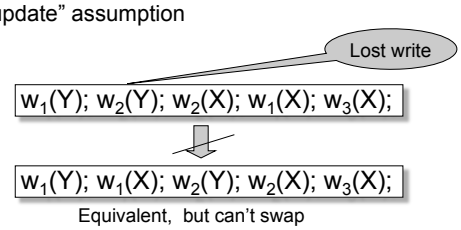
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

Conflict Serializability

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption



Equivalent, but can't swap

Scheduler

- The scheduler is the module that schedules the transaction's actions, ensuring serializability
- How ? We discuss three techniques in class:
  - Locks
  - Timestamps (next lecture)
  - Validation (next lecture)

Locking Scheduler

Simple idea:

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

Notation

$l_i(A)$ = transaction T_i acquires lock for element A

$u_i(A)$ = transaction T_i releases lock for element A

Example

T1	T2
$L_1(A)$; READ(A, t)	
t := t+100	
WRITE(A, t); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A,s)
	s := s*2
	WRITE(A,s); $U_2(A)$;
	$L_2(B)$; **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); $U_1(B)$;	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

Example

T1	T2
$L_1(A)$; READ(A, t)	
t := t+100	
WRITE(A, t); $U_1(A)$;	
	$L_2(A)$; READ(A,s)
	s := s*2
	WRITE(A,s); $U_2(A)$;
	$L_2(B)$; READ(B,s)
	s := s*2
	WRITE(B,s); $U_2(B)$;
$L_1(B)$; READ(B, t)	
t := t+100	
WRITE(B,t); $U_1(B)$;	

Locks did not enforce conflict-serializability !!!

Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must preceed all unlock requests
- This ensures conflict serializability ! (why?)

Example: 2PL transactions

T1	T2
$L_1(A)$; $L_1(B)$; READ(A, t)	
t := t+100	
WRITE(A, t); $U_1(A)$	
	$L_2(A)$; READ(A,s)
	s := s*2
	WRITE(A,s);
	$L_2(B)$; **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); $U_1(B)$;	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); $U_2(A)$; $U_2(B)$;

Now it is conflict-serializable

What about Aborts?

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?
- Serializable schedule definition only considers transactions that commit
  - Relies on assumptions that aborted transactions can be undone completely

Example with Abort

T1	T2
$L_1(A)$; $L_1(B)$; READ(A, t)	
t := t+100	
WRITE(A, t); $U_1(A)$	
	$L_2(A)$; READ(A,s)
	s := s*2
	WRITE(A,s);
	$L_2(B)$; **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); $U_1(B)$;	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); $U_2(A)$; $U_2(B)$;
Abort	Commit

Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed
- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit
- Avoids cascading rollbacks

Deadlock

- Transaction T_1 waits for a lock held by T_2;
- But T_2 waits for a lock held by T_3;
- While T_3 waits for . . . .
- . . .
- . . .and T_{73} waits for a lock held by T_1 !!
- A deadlock is when two or more transactions are waiting for each other to complete

Handling Deadlock

- **Deadlock avoidance**
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting
- **Deadlock detection**
  - Timeouts (but hard to pick the right threshold)
  - Wait-for graph
    - What commercial systems use (they check graph periodically)

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- U = update lock
  - Initially like S
  - Later may be upgraded to X
- I = increment lock (for A := A + something)
  - Increment operations commute

Recommended reading: chapter 18.4

Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- Coarse grain locking (e.g., tables)
  - Many false conflicts
  - Less overhead in managing locks
- Alternative techniques
  - Hierarchical locking (and intentional locks) [commercial DBMSs]
  - Lock escalation

Recommended reading: chapter 18.6

Phantom Problem

- A "phantom" is a tuple that is invisible during part of a transaction execution but not all of it.
- Example:
  - T0: reads list of books in catalog
  - T1: inserts a new book into the catalog
  - T2: reads list of books in catalog: New book appears!
- Can this occur?
- Depends on locking details
  - eg, granularity of locks
- To avoid phantoms needs **predicate locking**

The Locking Scheduler

Task 1:
Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure 2PL !

Recommended reading: chapter 18.5

The Locking Scheduler

Task 2:
Execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
  - Grant, or add the transaction to the element's wait list
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

Recommended reading: chapter 18.5

Degrees of Isolation

- Isolation level "serializable" (i.e. ACID)
  - Golden standard
  - Requires strict 2PL and predicate locking
  - But often too inefficient
  - Imagine there are only a few update operations and many long read operations
- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often **default**)
  - Sometimes are hard to understand

Degrees of Isolation

- **Four levels of isolation**
  - All levels use **long-duration exclusive locks**
  - READ UNCOMMITTED: no read locks
  - READ COMMITTED: short duration read locks
  - REPEATABLE READ:
    - Long duration read locks on individual items
  - SERIALIZABLE:
    - All locks long duration and lock predicates
- **Trade-off: consistency vs concurrency**
- Commercial systems give **choice** of level + **others**