Relational Database System Implementation

CS122 – Lecture 17

Winter Term, 2018-2019

Transaction Isolation

- ACID Properties:
  - Atomicity, Consistency, Isolation, Durability
- Have talked about atomicity, consistency, durability
  - Important whether the DB is single-user or multi-user
  - Traditional approach is to use a write-ahead log, although shadow-page technique shows up in some places
- Transaction isolation is *very* important when a DB can be used by multiple clients at the same time
  - Multiple concurrent operations against same data values
  - Without proper precautions, DB will produce spurious results
- Five kinds of spurious results can occur, without proper transaction isolation

Transaction Isolation Issues

- Dirty writes:
  - A transaction T_1 writes a value to X
  - Another transaction T_2 also writes a value to X, before T_1 commits or aborts
  - If T_1 or T_2 aborts, what should be the value of X?
- Dirty reads:
  - A transaction T_1 writes a value to X
  - T_2 reads X before T_1 commits
  - If T_1 aborts, T_2 has an invalid value for X
- Nonrepeatable reads:
  - T_1 reads X
  - T_2 writes to X, or deletes X, then commits
  - If T_1 reads X again, value is now different or gone

Transaction Isolation Issues (2)

- Phantom rows, a.k.a. phantoms:
  - Transaction T_1 reads rows that satisfy a predicate P
  - Transaction T_2 then writes rows, some of which satisfy P
  - If T_1 repeats its read, it gets a different set of results
  - If T_1 writes values based on original read, new rows aren't considered!
- Lost updates:
  - Transaction T_1 reads the value of X
  - Transaction T_2 writes a new value to X
  - T_1 writes to X based on its previously read value

Serial Transaction Execution

- A simple solution to transaction isolation issues:
  - Only allow one transaction to execute at a time
  - Transactions are executed in a *serial* order
- Problem: this is <u>really</u> slow
  - Doesn't maximize utilization of DB server resources
  - Transaction throughput will be really low
- Most of the time, transactions work with completely different records
- Isolation:
  - For every pair of transactions T_i and T_j, it *appears* that either T_i completes before T_j starts, or that T_j completes before T_i starts

Serializable Execution

- Most databases interleave transaction operations
  - As long as database is careful to maintain isolation, yields much higher transaction processing throughput
- Goal: ensure that transactions are executed in a way that is *equivalent to* a serial execution schedule
  - For every pair of transactions T_i and T_j, it *appears* that either T_i completes before T_j starts, or that T_j completes before T_i starts
- Called a *serializable* execution schedule
- Several different kinds of serializable schedules, with different characteristics
  - *Not all serializable schedules are created equal!*

SQL Isolation Levels

- Sometimes applications are immune to certain kinds of spurious results
  - e.g. nonrepeatable reads or phantom rows
  - App doesn't have queries that are affected by these behaviors, e.g. if most transactions are simple retrievals or updates
- Can define weaker forms of isolation
  - Weaker isolation allows greater concurrency, and therefore greater transaction throughput
  - Weaker isolation also allows more kinds of spurious results
- SQL defines four isolation levels for use in applications
  - Can set individual txns to have a specific isolation level

SQL Isolation Levels (2)

- Serializable:
  - Concurrent transactions produce the same result as if they were run in some serial order
  - The serial order may not necessarily correspond to the exact order that transactions were issued
- Called *strong isolation*
- Only level that satisfies original definition of isolation:
  - For every pair of transactions T_i and T_j, it appears that either T_i completes before T_j starts, or that T_j completes before T_i starts

SQL Isolation Levels (3)

- Other isolation levels are called *weak isolation*
  - Allow various kinds of spurious behavior in concurrent transactions
- Repeatable reads:
  - During a transaction, multiple reads of *X* produce same results, regardless of committed writes to *X* in other transactions
  - Other transactions' committed changes do not become visible in the middle of a transaction
  - (If the txn changes *X*, it sees its own modifications…)

SQL Isolation Levels (4)

- Read committed:
  - During a transaction, other transactions' committed changes become visible immediately
  - Value of *X* can change during a transaction, if other transactions write to *X* and then commit
- Read uncommitted:
  - Uncommitted changes to *X* in other transactions become visible immediately
- Aside: Many DBs also now include snapshot isolation
  - They sometimes call it serializable isolation, but it isn't!
  - Will discuss this isolation level in the future…

SQL Isolation Levels (5)

- Different SQL isolation levels allow different kinds of spurious behaviors:

Isolation Level	Dirty Writes	Dirty Reads	Nonrepeatable Reads	Phantoms
serializable	NO	NO	NO	NO
repeatable reads	NO	NO	NO	YES
read committed	NO	NO	YES	YES
read uncommitted	NO	YES	YES	YES

SQL Isolation Levels (6)

- Databases often allow clients to set the desired transaction isolation level
- SQL syntax:

```
SET TRANSACTION ISOLATION LEVEL
      [ SERIALIZABLE | REPEATABLE READS |
        READ COMMITTED | READ UNCOMMITTED ]
```

- Databases don't always support all isolation levels!
  - DB2, SQLServer, MySQL support all four isolation levels
  - Oracle and PostgreSQL only support SERIALIZABLE and READ COMMITTED isolation levels
    - (And by "serializable" isolation they mean "snapshot"…)

Transaction Schedules

- As before, model transactions as a series of reads and writes on various data items
- The sequences of operations that txns perform are called *schedules*
- A schedule can only perform <u>one</u> operation at a time
- A *serial* schedule never allows the reads and writes of two transactions to be interleaved
  - Very slow, but avoids all spurious results

T_i:
read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B);
commit.

T_j:
read(A);
$A := A - 30$;
write(A);
read(C);
$C := C + 30$;
write(C);
commit.

Serializable Schedules

- Want to interleave transaction operations to improve throughput
- Need to make sure that results are still valid
- Require that execution schedules are equivalent to a serial schedule
  - i.e. the schedule is *serializable*

T_i:	T_j:
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
read(B);	
$B := B + 50$;	
write(B);	
commit.	
	read(C);
	$C := C + 30$;
	write(C);
	commit.

- *What makes a schedule serializable?*
- *How do we know two schedules are equivalent?*

Serializable Schedules (2)

- Given:
  - A schedule *S* containing operations performed by two transactions, T_i and T_j
  - In the schedule, instruction *I* from transaction T_i is adjacent to instruction *J* from T_j
- In what situations can we swap the order of instructions *I* and *J* without affecting the results?
- If we cannot swap instructions *I* and *J* without affecting the results, we say that the operations *conflict*

Avoiding Conflicts

- A simple example:
  - Two adjacent instructions *I* and *J*, from different transactions T_i and T_j, respectively
  - Instruction *I* is read(*A*) or write(*A*), on a data-item *A*
  - Instruction *J* is read(*B*) or write(*B*), on a <u>different</u> data-item *B*
  - Does it matter what order we execute *I* and *J* ?

- If instructions *I* and *J* refer to different data-items, they do not conflict. We can execute *I* and *J* in any order.

Avoiding Conflicts (2)

- Instructions I and J could conflict if they read or write the same data item
- If I = read(Q) and J = read(Q), can we swap them without affecting the results?
  - Yes! Both transactions will see the same value for Q, regardless of the order.
- If I = read(Q) and J = write(Q), can we swap them without affecting the results?
  - No! If I executes before J, T_i will see the old value of Q. If I executes after J, T_i will see the new value of Q.

Avoiding Conflicts (3)

- Same issue if I = write(Q) and J = read(Q)
  - Cannot swap order of I and J without affecting the results
- If I = write(Q) and J = write(Q), can we swap them without affecting the results?
  - The write operations themselves will not be affected…
  - …but the next read of Q will see different results based on the order of I and J
  - Cannot swap order of I and J without affecting the results

Conflict Equivalence

- Given a transaction execution schedule *S*, with two adjacent operations *I* and *J* from different transactions
- Instructions *I* and *J* conflict if:
  - *I* and *J* operate on the same data item
  - At least one of the operations is a write
- If the instructions *I* and *J* do not conflict:
  - We can swap them to produce an equivalent schedule *S'*
  - Execution of *S* or *S'* will produce the exact same results

Conflict Equivalence (2)

- A pair of schedules *S* and *S'* are *conflict equivalent* if:
  - One schedule can be transformed into the other, solely by swapping adjacent non-conflicting operations

- A schedule *S* is *conflict serializable* if it is conflict equivalent to a serial schedule

Previous Example Schedules

- Are these schedules conflict equivalent?
- Yes: only non-conflicting operations are swapped.

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
read(B);	
$B := B + 50$;	
write(B);	
	read(C);
	$C := C + 30$;
	write(C);
commit.	
	commit.

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
read(B);	
$B := B + 50$;	
write(B);	
commit.	
	read(C);
	$C := C + 30$;
	write(C);
	commit.

Previous Example Schedules (2)

- Is the right schedule conflict serializable?
- Yes! Left schedule is a serial execution schedule.

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
read(B);	
$B := B + 50$;	
write(B);	
commit.	
	read(A);
	$A := A - 30$;
	write(A);
	read(C);
	$C := C + 30$;
	write(C);
	commit.

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
read(B);	
$B := B + 50$;	
write(B);	
commit.	
	read(C);
	$C := C + 30$;
	write(C);
	commit.

Another Example

- Again, is the right schedule conflict serializable?
- Yes. Left schedule is serial; right is conflict equivalent.

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
read(B);	
$B := B + 50$;	
write(B);	
commit.	
	read(A);
	$A := A - 30$;
	write(A);
	read(C);
	$C := C + 30$;
	write(C);
	commit.

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
	read(C);
	$C := C + 30$;
	write(C);
	commit.
read(B);	
$B := B + 50$;	
write(B);	
commit.	

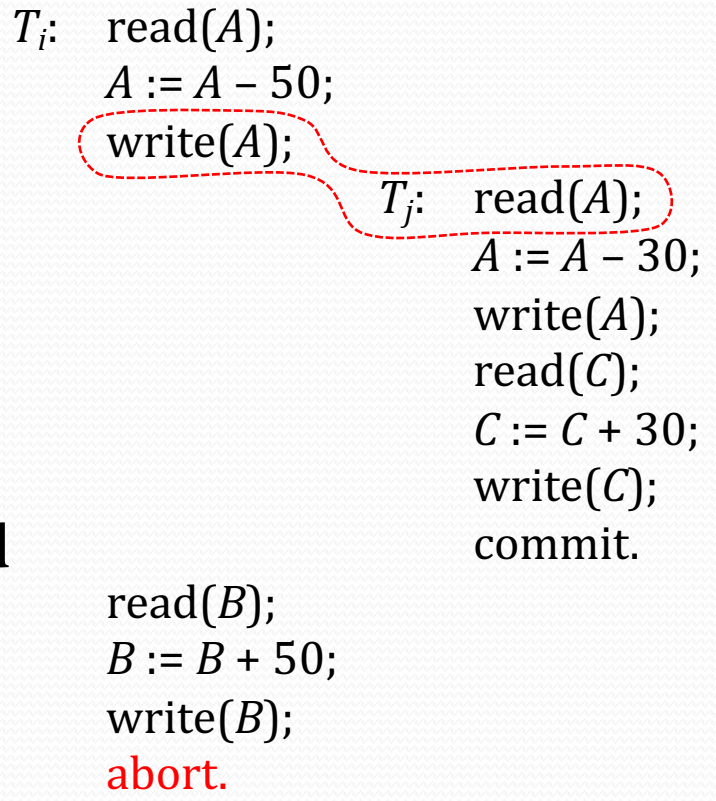
A New Problem

- What if we have this execution schedule, but T_i aborts?
- This execution schedule violates atomicity!
  - T_i modifies data-item A…
  - Then T_j also modifies A, based on T_i's changes!
  - Then T_j commits, preserving T_i's changes to A, even though T_i is eventually aborted.
- Could preserve atomicity by aborting T_j when T_i aborts…
  - That would violate durability!!

T_i	T_j
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
	read(C);
	$C := C + 30$;
	write(C);
	commit.
read(B);	
$B := B + 50$;	
write(B);	
abort.	

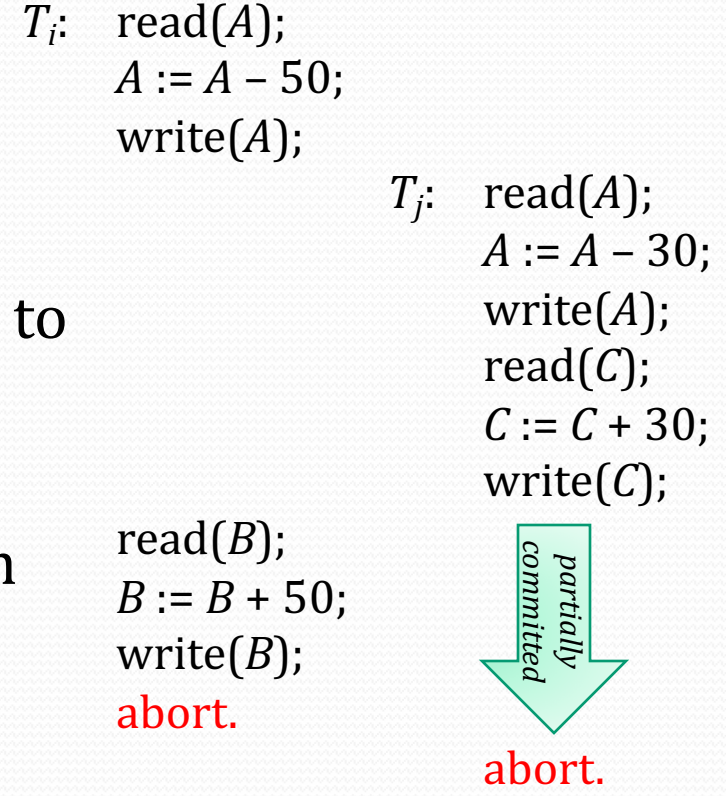
Recoverable Schedules

- This is a *nonrecoverable* execution schedule
  - Can't properly enforce atomicity, consistency and durability with this schedule
- Want to constrain ourselves to only *recoverable schedules*
- A schedule S is recoverable if, for every pair of txns T_i and T_j:
  - If T_j reads a data-item previously written by T_i, then T_j is not allowed to commit until T_i first commits



Recoverable Schedules (2)

- Can make this schedule recoverable simply by delaying T_j's commit operation
  - T_j enters "partially committed" state initially
  - When T_i aborts, T_j is also aborted
- Transaction T_j is *dependent* on T_i:
  - T_j reads a value that T_i has written to
- General rule:
  - Dependent transactions may not commit until the initial transaction commits



Recoverable Schedules (3)

- If T_i aborts then we must abort T_j too
  - Called a *cascading rollback*
- This can get very expensive
  - Very easy to introduce dependencies between interleaved transactions
  - Aborting one transaction can cause a large amount of work to be discarded

T_i:	T_j:	T_k:
read(A);		
$A := A - 50$;		
write(A);		
	read(A);	
	$A := A - 30$;	
	write(A);	
	read(C);	
	$C := C + 30$;	
	write(C);	
		read(C);
		$C := C * 1.03$;
		write(C);
read(B);		
$B := B + 50$;		
write(B);		
abort.		
	abort.	
		abort.

Cascadeless Schedules

- *Cascadeless schedules* disallow cascading rollbacks from occurring
- A schedule S is cascadeless if, for every pair of transactions T_i and T_j:
  - If T_j reads a data-item previously written by T_i, then T_j is not allowed to perform this read until T_i first completes
- Question:
  - What if T_j writes (but never reads) a data-item that T_i previously wrote, and then T_i is aborted?
  - Don't need to cascade the rollback to T_j …
  - T_j never saw the old value!

Blind Writes

- One more example:
  - Original value of A is 1
- All writes in these transactions are *blind writes*
  - The data item is not read before it is written
- Is this schedule cascadeless?
  - This schedule is cascadeless
  - T_j doesn't read A at all

T_i:	A := 2		
	write(A);		
		T_j:	A := 3
			write(A);
	abort.		
			abort.

Blind Writes (2)

- As these txns are executed, write-ahead log is updated
- When T_i aborts, it issues a compensating log record, as usual
- When T_j aborts, it also issues a compensating log record…
  - …except that the old value of A is from an aborted transaction
- Original value of A was 1, but after T_i and T_j are aborted, it's 2 ☹

T_i:	T_j:
A := 2	
write(A);	
	A := 3
	write(A);
abort.	
	abort.

Write-Ahead Log:

T_i: start
T_i: A, 1, 2
T_j: start
T_j: A, 2, 3
T_i CLR: A, 1
T_i: abort
T_j CLR: A, 2
T_j: abort

Blind Writes (3)

- Problem:
  - The before-value of A written to the write-ahead log for T_j was taken from the *incomplete* transaction T_i
  - If T_i aborts, the before-value T_j has is <u>useless</u> for rollback!
- Writes to a data-item also introduce subtle dependencies between txns, *through the write-ahead log!*

T_i:		T_j:
	A := 2	
	write(A);	
		A := 3
		write(A);
	abort.	
		abort.

Write-Ahead Log:

T_i: start
T_i: A, 1, 2
T_j: start
T_j: A, 2, 3
T_i CLR: A, 1
T_i: abort
T_j CLR: A, 2
T_j: abort

Strict Schedules

- To simplify recovery processing, further constrain transaction schedules to be strict
- A schedule *S* is *strict* if, for every pair of txns T_i and T_j:
  - If T_j reads or writes a data-item previously written by T_i, then T_j is not allowed to do this until T_i first commits
- If the database only uses strict execution schedules:
  - When a transaction first writes to any given data-item, the update record written to the WAL will never contain an uncommitted value from another transaction
  - This makes recovery processing much simpler

Transaction Schedule Hierarchy

- Can subdivide space of transaction schedules based on classifications discussed today: