Transaction Management: Concurrency Control

Abdu Alawini

University of Illinois at Urbana-Champaign

CS411: Database Systems

October 17, 2018

Some slides used with permission from David Maier, Susan Davidson and Lois Delcambre

Announcements

- Project Track 1 (PT1) – Stage 3 is due TODAY
- Please sign up for PT1 midterm demos asap
- **Midterm: 10/29 in class 11-12:15 pm**
- **Midterm review session: Friday 10/26 4:00-4:50 at SC 1404**
  - **Don’t forget to fill the form with topics to discuss in the review session**

Announcements, cont.

- Topics covered on the midterm
  - SQL: DDL, DML, null values, constraints, views
  - ER\UML modeling, translation to relations
  - Relational design theory: FDs and BCNF, 3NF decomposition.
  - Transactions: 2-phase locking, isolation levels

Recap: Transaction

A transaction is:

- one “complete” set of actions
- defined by the user (meaningful to the application)
- establishes where certain integrity constraints are enforced.

For concurrency control purposes (inside DBMS):

- a transaction is one atomic unit of work.

  Thus we must be able to undo it if incomplete
- DBMS cares only about the reads/writes to the DB
- DBMS views a transaction as (only) a sequence of reads, writes plus commit & abort
  (mostly ignores the actual SQL that generated the reads and writes)

Today's lecture

- Transactions and SQL: isolation levels

- How the database implements isolation levels
- Theory of serializability

Characterizing Conflicts

T_1 T_2

$R_1(S)$

$R(S)$

$W_1(S)$

$W(S)$

- Two consistency preserving, committed transactions T_1 and T_2 can run against a consistent database and leave it in an inconsistent state due to three types of conflicts:
  - Read-Write (RW): T_1 reads a data object that is subsequently written by T_2
  - Write-Read (WR): T_1 writes a data object that is subsequently read by T_2
  - Write-Write (WW): T_1 writes a data object that is also subsequently written by T_2

WR

Dirty read

What were the conflicts for the previous executions?

WR Conflicts: Dirty Reads

- Dirty data is data written by an uncommitted transaction; a dirty read is a read of dirty data.

- Sometimes we can tolerate dirty reads; other times we cannot.

SQL query: computes the avg age of male & female in a company

Example of Dirty Reads

sid	name	zip
123	Qin	14001
321	Kristin	14104

Transaction 1	Transaction 2
BEGIN;	BEGIN;

	UPDATE student
	SET zip = 14111 WHERE sid = 123;

SELECT zip FROM student WHERE sid = 123; /* will read 14111*/	
	ROLLBACK /* zip code will revert to 14001*/

Other Undesirable Phenomena: Unrepeatable read (RW conflict)

- Unrepeatable read (RW conflict): a transaction reads the same data item twice and gets different values
  - The airline seat reservation is an example of where an unrepeatable read could occur.

Example of Non-repeatable Reads

sid	name	zip
123	Qin	~~14001~~ 14111
321	Kristin	14104

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT zip FROM student WHERE sid = 123; /* will read 14001*/	

	UPDATE student
	SET zip = 14111 WHERE sid = 123;

	COMMIT;
SELECT zip FROM student WHERE sid = 123; /* will read 14111*/	

Other Undesirable Phenomena:
Overwriting uncommitted data (WW conflict)

- Overwriting uncommitted data (WW conflict): one transaction overwrites the value of data that is in the process of being written by another transaction
  - The flight seat selection we've seen earlier is an example of WW conflict.

T_1 T_2

$R(S)$

$R(S)$

$W(S)$

$W(S)$

Phantom Reads

- A phantom read occurs when, in the course of a transaction, new rows are added by another transaction to the records being read.*

* https://en.wikipedia.org/wiki/Isolation_(database_systems)

Example of Phantom Reads

	sid	name	zip
1	123	Qin	14001
2	321	Kristin	14104
3	100	Morgan	14444

Transaction 1	Transaction 2
BEGIN;	BEGIN;
4 SELECT COUNT(*) FROM students WHERE sid BETWEEN 100 AND 400; /* will return 2 */ 2	

	INSERT INTO students(sid,name,zip) VALUES (100, ‘Morgan’, 14444);
	COMMIT;

SELECT COUNT(*) FROM students WHERE sid BETWEEN 100 AND 400; /* will return 3 */	

Isolation

- The problems we've seen are all related to *isolation*
- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is more expensive than "no isolation"
    - We can't do as many things concurrently (or we have to undo them frequently)
  - For performance, we generally want to specify the most relaxed *isolation level* that's acceptable
    - Note that we're "slightly" violating a correctness constraint to get performance!

Specifying Acceptable Isolation Levels

- To signal to the system that a dirty read is acceptable;

  SET TRANSACTION READ ONLY

  ISOLATION LEVEL READ UNCOMMITTED;

DR , UR

PR

- In addition, there are

  SET TRANSACTION

  ISOLATION LEVEL READ COMMITTED;

UR, PR

  SET TRANSACTION

  ISOLATION LEVEL REPEATABLE READ;

PR

READ COMMITTED

- Forbids the reading of dirty (uncommitted) data, but allows a transaction T to *issue the same query several times and get different answers*
  - No value written by T can be modified until T completes
- For example, the Reservation example could also be READ COMMITTED; the transaction could repeatably poll to see if the seat was available, hoping for a cancellation

REPEATABLE READ

- What it is NOT: a guarantee that the same query will get the same answer!
- However, if a tuple is retrieved once it will be retrieved again if the query is repeated
  - For example, suppose Reservation were modified to retrieve all available seats
  - If a tuple were retrieved once, it would be retrieved again (but additional seats may also become available)

Summary of Isolation Levels

Level	Dirty Read	Unrepeatable Read	Phantoms
READ UN-COMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Outline

- ✓ Transactions and SQL: isolation levels

- How the database implements isolation levels
- Theory of serializability

Implementing Isolation Levels

- One approach – use locking at some level (tuple, page, table, etc.):
  - each data item is either locked (in some mode, e.g. shared or exclusive) or is available (no lock)
  - an action on a data item can be executed if the transaction holds an appropriate lock
  - consider *granularity* of locks – how big of an item to lock
    - Larger granularity = fewer locking operations but more contention!

- Appropriate locks:
  - Before a read, a shared lock must be acquired
  - Before a write, an exclusive lock must be acquired

S(A) R(A)
X(A) W(A)

Lock Compatibility Matrix

Locks on a data item are granted based on a lock compatibility matrix:

		Mode of Data Item		
		None	Shared	Exclusive
Request mode	Shared	Y	Y	N
	Exclusive	Y	N	N

When a transaction requests a lock, it must wait (block) until the lock is granted

$W_1(A)$ $W_2(A)$

$X_1(A)$ $W_1(A)$ $Rel(A)$

$X_2(A)$ < Wait >

Locks Prevent "Bad" Execution

If the system used locking, the first "bad" execution could have been avoided: X = Seat 22A

User 1	User 2
xlock(X)	x(X)
read(X.avail)	
IF (!X. avail)	{xlock(X) is not granted}
{X. avail := 1}	
write(X. avail)	
release(X)	
	xlock(X)
	read(X.avail)
	IF(!X. avail)
	{X. avail := 1}
	write(X. avail)
	release(X)

Locks are not enough...

User 1	User 2
slock(X)	
read(X.avail)	
	slock(X)
	read(X.avail)
	release(X)
release(X)	
	IF (X.avail)
	{X.avail := 1}
IF (!X.avail)	
{X.avail := 1}	
xlock(X)	
write(X.avail)	
release(X)	
	xlock(X)
	write(X.bal)
	release(X)

Isolation Levels and Locking

READ UNCOMMITTED allows queries in the transaction to read data *without acquiring any lock*

- Access mode READ ONLY, no updates are allowed

READ COMMITTED requires a read-lock to be obtained for all tuples touched by queries, but it releases the locks immediately after the read

- Exclusive locks must be obtained for updates and held to end of transaction

Isolation levels and locking, cont.

PR

REPEATABLE READ places shared locks on tuples retrieved by queries, holds them until the end of the transaction

- Exclusive locks must be obtained for updates and held to end of transaction

SERIALIZABLE places shared locks on tuples retrieved by queries *as well as the index*, holds them until the end of the transaction

- Exclusive locks must be obtained for updates and held to end of transaction

Holding locks to the end of a transaction is called "strict" locking

Putting it all together….

- Suppose we have two transactions:

T1: SET TRANSACTION READ WRITE
ISOLATION LEVEL READ COMMITTED;

SQL code that translates to: R1(A), R1(B) W1(B) W1(C)

T2: SET TRANSACTION READ WRITE
ISOLATION LEVEL READ COMMITTED;

SQL code that translates to: R2(C), R2(A) W2(A)

One possible interleaved execution of the transactions above:

R1(A) R2(C) R2(A) W2(A) R1(B) W1(B) W1(C)

S1(A) R1(A) REL1(A) S2(C) R2(C) REL2(C) X2(A) R2(A) W2(A) X1(B) R1(B) W1(B) X1(C) W1(C) REL1(B,C) REL2(A)

Putting it all together….

A	B	C
S_1	X_1	

- Now suppose that T1 is REPEATABLE READ:

T1: SET TRANSACTION READ WRITE

ISOLATION LEVEL REPEATABLE READ;

SQL code that translates to: R1(A), R1(B) W1(B) W1(C)

T2: SET TRANSACTION READ WRITE

ISOLATION LEVEL READ COMMITTED;

SQL code that translates to: R2(C), R2(A) W2(A)

One possible interleaved execution of the transactions above:

Corrected

R1(A) R2(C) R2(A) R1(B) W1(B) W1(C) R2(A) W2(A)

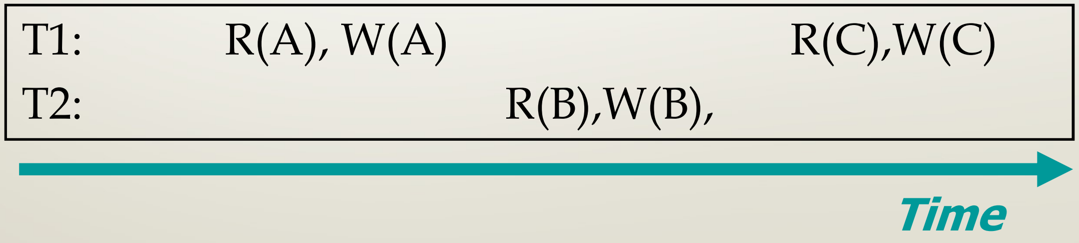
S1(A) R1(A) S2(C) R2(C) REL2(C) X2(A)<T2 must wait until the lock on A is released> X1(B) R1(B) W1(B) X1(C) W1(C) REL1(A, B,C) X2(A) R2(A) W2(A) REL2(A)

Outline

- ✓ Transactions and SQL: isolation levels
- ✓ How the database implements isolation levels
- ● Theory of serializability

Schedules

- Schedule: An interleaving of actions from a set of transactions, where the actions of each individual transaction are in the original order.
  - Represents an actual sequence of database actions.
  - Example: $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, $R_1(C)$, $W_1(C)$
  - In a *complete* schedule, each transaction ends in commit or abort.
- Initial State of DB + Schedule → Final State of DB



T1:	R(A), W(A)		R(C),W(C)
T2:		R(B),W(B),	

Serializable Schedule ⇔ Isolated Transactions

$T_2 \rightarrow T_1$

$T_1 \rightarrow T_2$

- Serial schedules:

  - Run transactions one at a time, in a series. (Different orders might give different results.)

- Serializable schedules:

  - Final state must be the same as the state produced by one of the serial schedules.
  - Must appear to each transaction as if the transactions that precede it ran sequentially

$T_1 \rightarrow T_2$

$R_1(A)$ $W_1(A)$

$R_2(B)$ $W_2(B)$

Questions to Address

- Given a schedule S, is it serializable?

- Two schedules are conflict-equivalent if we can convert one into the other by a sequence of nonconflicting swaps of adjacent actions

- How can we "restrict" transactions in progress to guarantee that only serializable schedules are produced?

$T_1 \to T_2$

Examples on Serializable Schedules

Which of these schedules is serializable?

$T_1 \to T_2$ T_1 T_2

R(A) W(A) R(B) W(B)

S1

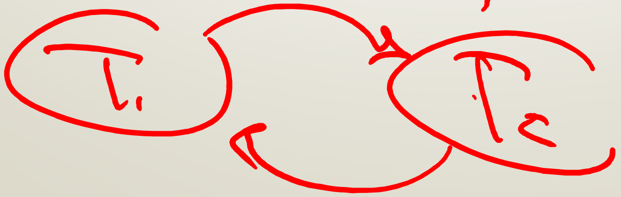
T1:	R(A), W(A),		R(B), W(B)	
T2:		R(A),W(A)		R(B), W(B)

✓

S2

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A),W(A) R(B),W(B)	

X



Conflict-Serializability

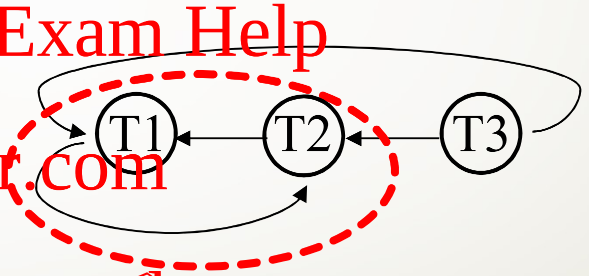
- A schedule is *conflict-serializable* if it is conflict-equivalent to a serial schedule.
- a conflict-serializable schedule is a serializable schedule
  - A schedule may be serializable but not conflict-serializable (read page 893)
- Conflict-serializability is a condition that the schedulers in commercial systems generally use when they need to guarantee serializability.

Testing for Conflict-Serializability

- Given a schedule S, we can construct a directed graph G=(V,E) called a precedence graph
  - V : all transactions in S
  - E : $T_i \rightarrow T_j$ whenever an action of T_i precedes and conflicts with an action of T_j in S (RW, WR, WW)
- Theorem:
  A schedule S is conflict serializable if and only if its precedence graph contains no cycles
  - Note that testing for a cycle in a digraph can be done in time $O(|V|^2)$

An Example

T1	T2	T3
		R(X,Y,Z)
R(X)		
W(X)		
	R(Y)	
	W(Y)	
R(Y)		
	R(X)	
		R(Z)



Cyclic: Not conflict-serializable.