Transactions

Serializability

Isolation Levels

Atomicity

1

The Setting

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike Operating Systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

2

Example: Bad Interaction

- You and your spouse each take $100 from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.
- Compare: An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

3

ACID Test

- A DBMS is expected to support "ACID transactions," which are:
  - *Atomic* : Either the whole process is done or none is.
  - *Consistent* : Database constraints are preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time. (aka "serializable")
  - *Durable* : Effects of a process do not get lost if the system crashes.

4

Transactions in SQL

- SQL supports transactions, often behind the scenes.
  - Each statement issued at the generic query interface is a transaction by itself.
  - In programming interfaces like Embedded SQL or PSM, a transaction begins the first time an SQL statement is executed and ends with the program or an explicit end.

5

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - Its database modifications are now permanent in the database.

6

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 can also cause rollback, even if the programmer does not request it.

7

An Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

8

Sally’s Program

- Sally executes the following two SQL statements, which we call (min) and (max), to help remember what they do.

(max) SELECT MAX(price) FROM Sells
WHERE bar = ‘Joe’’s Bar’;

(min) SELECT MIN(price) FROM Sells
WHERE bar = ‘Joe’’s Bar’;

9

Joe’s Program

- At about the same time, Joe executes the following steps, which have the mnemonic names (del) and (ins).

(del) DELETE FROM Sells
WHERE bar = ‘Joe’’s Bar’;

(ins) INSERT INTO Sells
VALUES(‘Joe’’s Bar’, ‘Heineken’,
3.50);

10

Interleaving of Statements

- Although (max) must come before (min) and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe's Prices:	2.50, 3.00	2.50, 3.00		3.50
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

Fixing the Problem With Transactions

- If we group Sally's statements (max)(min) into one **transaction**, then she cannot see this inconsistency.
- She see's Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

13

Another Problem: Rollback

- Suppose Joe executes (del)(ins), but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her transaction after (ins) but before the rollback, she sees a value, 3.50, that **never** existed in the database.

14

Solution

- If Joe executes (del)(ins) as a **transaction**, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can ***never*** be seen.

15

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.

- How a DBMS implements these isolation levels is highly complex, and a typical DBMS provides its own options.

16

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE (strongest)
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED (weakest, effectively unrestricted)

17

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.
- It's up to the DBMS vendor to figure out how to do that, e.g.:
  - True isolation in time.
  - Keep Joe's old prices around to answer Sally's queries.

18

Isolation Level Is "Personal" Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it.
- Example: If Joe Runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

19

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can **see only committed** data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees MAX < MIN.

20

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then **everything seen the first time will be seen the second time**.
  - **But** the second and subsequent reads may see *more* tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

23

Locking

- Locking is a system-level method for achieving serialization.
- The diagrams in slides on locking are modifications of ones due to Arthur Keller, rather than Jeff Ullman, I believe.

24

Non-Serialized Transaction

T1	T2	**start with A = 5**		
Read A		A on disk	A in T1	A in T2
	Read A	5	5	5
A:= A + 1		5	6	5
	A:= 2* A	5	6	10
	Write A	10	6	10
Write A		6	6	10

25

Read- vs. Write-Locks

RLOCK A
WLOCK A
UNLOCK A

		THEM		
		NO	R	W
US	NO	OK	OK	OK
	R	OK	OK	bad
	W	OK	bad	bad

RLOCK –> UNLOCK can enclose a read
WLOCK –> UNLOCK can enclose a write or read

26

Transaction Serialized using Locks

T1	T2
WLOCK A	
Read A	
	WLOCK A
A:= A+1	
Write A	waits
UNLOCK A	
	granted
	Read A
	A:=2*A
	Write A
	UNLOCK A

27

Problems with Unstructured Use of Locks

T1	T2	
RLOCK A		
Read A		
	RLOCK A	
	Read A	
A:= A+1		
	A:= 2*A	
	WLOCK A	request lock upgrade
WLOCK A		request lock upgrade
wait	waits	

Deadlock!

28

Deadlock due to lock ordering

T1	T2
WLOCK A	
	WLOCK B
WLOCK B	
wait	WLOCK A
UNLOCK A	wait <u>deadlock</u>
	UNLOCK B
UNLOCK B	
	UNLOCK A

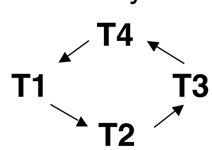
29

Deadlock

1. Wait and hold hold some locks while you wait for others

2. Circular chain of waiters

wait-for graph



3. No pre-emption

We can **avoid** deadlock by doing at least ONE of:

1. Get all your locks at once
2. Apply an ordering to acquiring locks
3. Allow preemption (for example, use timeout on waits)

30

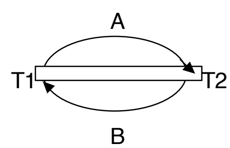
Serializability of schedules

T1	T2
Read (A)	Read (A)
A:= A-50	temp:= A * 0.1
Write (A)	A:= A + temp
Read (B)	Write (A)
B:= B+50	Read (B)
Write (B)	B:= B - temp
	Write (B)

	A	B
disk	100	200
T1		
T2		

A schedule is serializable iff its effect is the same as ***some*** serial schedule

T1 –> T2	T2 –> T1
A=	A=
B=	B=



31

Ignore the exact operation semantics (too fine-grain, and generally will lead to undecidability).
What is important is the same **sequence** of operations.

Conflicts
- Read-write
- Write-write

Two schedules S1, S2 are **equivalent** if
1. Each transaction does the same operations in S1 and S2.
2. For each data item Q,
   if in S1, Ti executes Read (Q)
   and the value of Q read by Ti was written by Tj
   then the same is true in S2.
3. For each data item Q,
   if in S1, transaction Ti executes the **last** write (Q),
   then same is true in S2

A schedule is serializable iff it is **equivalent** to some serial schedule.

32

Wanted:

A way to test serializabilty
by analyzing lock operations
among transactions

33

Serializability Test

Algorithm: Testing serializability of a schedule

Input: Schedule S for **transactions** T1,… , Tk

Output: Determination of whether S is serializable,
and if so, an equivalent serial schedule.

Method: Create a directed graph G (called a **serialization graph**)
Create a node for each transaction and label with transaction ID
Create an edge for each Ti: UNLOCK A followed by Tj: LOCK A
(where lock modes conflict). (A is the data item being locked.)
Label edge Ti --> Tj with A.

If there is a cycle then schedule is non-serializable.

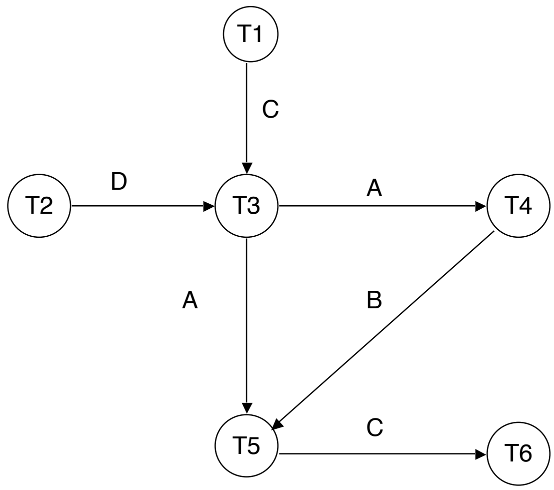
If there is no cycle, then (it is a DAG)
do a **topological sort** to get a serial schedule

DAG implies some partial order.

Any total order **consistent** with the partial order is an equivalent serial schedule.

34

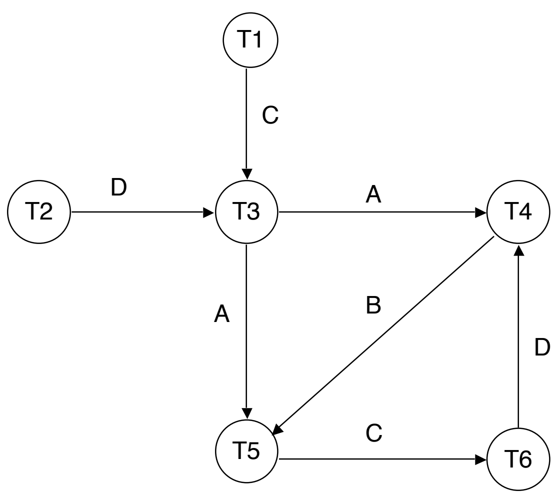
Serializability Test Example 1



T1
T2
T3
T4
T5
T6

35

Serializability Test Example 2

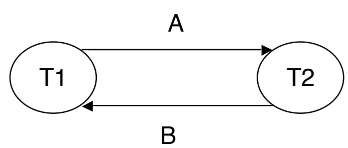


T1
T2
T3
?

36

Simplest Non-Serializable Case

T1	T2
WLOCK A	
UNLOCK A	
	WLOCK A
	UNLOCK A
	WLOCK B
	UNLOCK B
WLOCK B	
UNLOCK B	



Possible Serialization Strategies

- Static transaction analysis
- Dynamic detection of non-serializability, rollback if necessary
- Devise a safe but simple policy that avoids non-serializability

Simple Strategy: 2-Phase Locking (2PL)

Every transaction has is divided into two phases that occur in sequence:

Phase I: All requesting of locks (no releasing)
Phase II: All releasing of locks (no further requesting)

Theorem: Any schedule for 2-phase locked transaction is serializable

[Similar-sounding, but distinct idea: **2-Phase Commit** used in distributed databases.]

39

Rollback Hazards

T1	**T2**
LOCK A	
Read A	
change A	
Write A	
UNLOCK A	
	LOCK A
	Read A
	change A
	Write A
	UNLOCK A
LOCK B	
Read B	
Discover problem	
ABORT: ROLLBACK	
	Need to undo the change to A by T2 that was made based on data written by T1. (CASCADED ROLLBACK)

40