Transactions

CSC 321/621 - 03/29/2012

Transactions: Motivation

- At this point, you should have the notion that
  - MySQL is a server, which clients can connect to query data from and modify data in various databases
  - Client<u>s</u> is plural, meaning more than one user may be dealing with the data at the same time

- It is a responsibility of the DBMS system to provide a mechanism to ensure that all users see a sane and consistent view of the database

Transactions: Motivation

- The interleaving of actions between clients on the database can lead to problems

Which of these should we be concerned about?

- *Multiple reads?*
- *Reads and write?*
- *Multiple writes?*

Transactions: Motivating Example

- A “this-is-important” example:
  - You and your spouse have \$100.00 in your account. You are at two separate ATMs, each withdrawing \$60.00.
  - An ATM must
    - Check you have enough balance to withdraw the requested amount (a read)
    - Reduce the balance read previously by the withdrawn amount and write that back as the new balance (a write).
  - This scenario should end in:
    - One of you takes out \$60.00, the other is declined for going over the limit.

Transactions: Motivating Example

Assume you are user1, performing R1, W1 and your spouse is user2, performing R2, W2

(R1 = read1…)

What are the possible outcomes given possible interleavings?

Transactions: Motivating Example

Orders: You (1), Spouse (2)
(all orderings must have R1 < W1, R2 < W2)

R1, W1, R2, W2: You get \$60, spouse denied (OK)
R1, R2, W1, W2: You get \$60, spouse gets \$60, bank loses (NOT OK)
R1, R2, W2, W1: You get \$60, spouse gets \$60, bank loses (NOT OK)
R2, W2, R1, W1: Spouse gets \$60, you denied (OK)
R2, R1, W2, W1: You get \$60, spouse gets \$60, bank loses (NOT OK)
R2, R1, W1, W2: You get \$60, spouse gets \$60, bank loses (NOT OK)

Transactions: Motivating Example

OK cases: All of your database accesses occur together, and all of your spouses occur together

R1, W1, R2, W2

R2, W2, R1, W1

Not OK cases: Your and your spouses are interleaved

R1, R2, W1, W2

R1, R2, W2, W1

R2, R1, W2, W1

R2, R1, W1, W2

Transactions: Definition

- It is in the bank's best interest (and yours, as the bank's DBA) to ensure that the actions required for a given ATM withdrawal (R,W) are isolated from other database reads and writes

- **Transaction:** An action, or series of actions, carried out by a single user or application program that reads or updates the contents of the database and should be considered a *single logical unit of work.*

Transactions: Motivating Example

- Imagine we delete an employee from a database, and we have a foreign key from another table referencing that employee
  - If there is a CASCADE DELETE rule, then all tuples in that other table must be deleted that reference the target employee
  - This is a significant number of changes to the database, but should considered as one unit of work whose effects should not be visible until complete

Transactions: Concepts

- A transaction should move the database from a consistent state to another consistent state once completed
  - While executing, it may be inconsistent, but the result of the transaction must ensure consistency
  - Consistency includes ensuring no integrity constraints are violated

- It is up to the developer programming against the database to indicate what defines a transaction
  - We'll come back to this

Transactions: Terminology

- If a transaction completes, it should be *committed* (made permanent)
- If a transaction that needs to be aborted, it should be *rolled back (undone), with* all changes made to the database since the start of the transaction removed.
  - Ensures consistency by going back to state before transaction, which was consistent.
- There is special syntax for indicating to COMMIT or ROLLBACK a transaction as well

Transactions: ACID Properties

- Databases should support four properties of transactions: ACID PROPERTIES
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- We'll go into detail of each of these on the next slides

Transactions: Atomicity

- **Atomicity:** A transaction is an *all or nothing* event.

- Transactions are indivisible and must be completed in their entirety or not at all
  – Rollback facilities help here to get back to “not at all” if needed

Transactions: Consistency

Already discussed this

- **Consistency:** A transaction should move the database from a consistent state to another consistent state once completed
  - While executing, it may be inconsistent, but the result of the transaction must ensure consistency
  - Consistency includes ensuring no integrity constraints are violated

Transactions: Isolation

- **Isolation:** Transactions should occur independently of each other.

- Partial effects should not be visible to others.

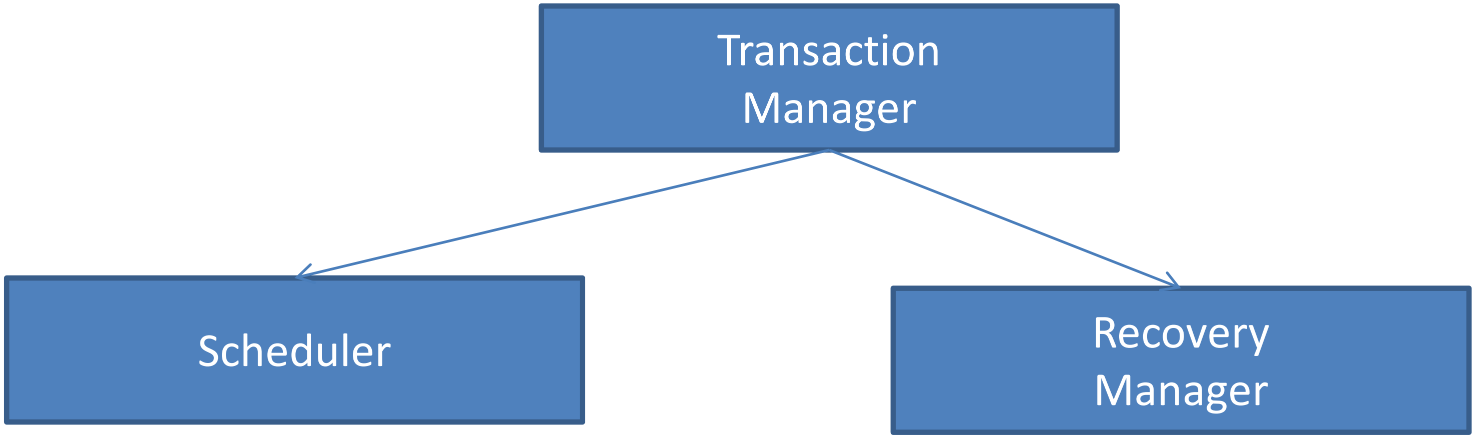
- Requires concurrency management.

Transactions: Durable

- **Durable:** A committed transaction should be considered as permanently recorded in the database and should be recoverable from subsequent failure.
  – This is a whole different can of worms, which we'll cover later!

Transactions: Database Support

- A DBMS typically supports transactions through the interactions of three components of the DBMS:



Implements concurrency controls

Supports rollbacks to previous states

Transactions: Concurrency Control

- Concurrency control is the management of database operations requested by multiple users

- Already discussed:
  - Reads that overlap: OK, no problems
  - Reads and writes: Problematic
  - Writes and writes: Problematic

Transactions: Concurrency Control

- Three common potential problems:
  - Lost update problem
  - Uncommitted dependency problem
  - Inconsistent analysis problem

- Let’s look at each of these in depth…

Transactions: Lost Update Problem

- **Lost Update:** When one update to the system is overwritten by another update to the system
- Back to our ATM example: One user depositing \$100.00, another user withdrawing \$10.00

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

- A solution: Force T1 read after T2 write (at time t6 or later)?

Transactions: Uncommitted Dependency Problem

- **Uncommitted Dependency:** Occurs when one transaction is allowed to see intermediate results of another transaction before commitment (and the "another" is then aborted)
  - *AKA Dirty Read*

Time	T_3	T_4	$\mathbf{bal_x}$
t_1		begin_transaction	100
t_2		read($\mathbf{bal_x}$)	100
t_3		$\mathbf{bal_x} = \mathbf{bal_x} + 100$	100
t_4	begin_transaction	write($\mathbf{bal_x}$)	200
t_5	read($\mathbf{bal_x}$)	⋮	200
t_6	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	rollback	100
t_7	write($\mathbf{bal_x}$)		190
t_8	commit		190

Transactions: Inconsistent Analysis Problem

- **Inconsistent Analysis:** When several values are being read from the db by one transaction and another transaction is updating some of those values
- Example: aggregate function applied on table while table being written (aggregate = summing in this example)

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Transactions: Managing Concurrency

- All of these problems should be fixable by forcing transactions to run sequentially and separately, one after another.

- It is possible to automate concurrency management, interleaving transaction operations sometimes?

Transactions: Locking

- To help enforce separation of access to data, the notion of *locks* on data has been developed and is the most common way to support concurrency.
  - Shared/Read lock:
    - If a transaction has a shared/read lock on a data item, it can read the item but not update it.
    - More than one transaction can have a shared/read lock on a data item (remember, read overlaps are fine)
  - Exclusive/Write lock :
    - If a transaction has an exclusive/write lock on a data item, it can read and update the item.
      - Updates often require a read and a write of the same item (think j = j + 10)
    - An exclusive/write lock is only provided to one transaction at a time.

Transactions: Locking

- We assume that locks can be *explicitly* requested and released, as well as implicitly released if a *rollback* or *commit* occurs.

- Lock implementation:
  - Extra bits associated with a data representing locked/unlocked and type of lock (read/write)
  - A separate table recording locked items & type of lock
  - As well as records of requested locks and types

Transactions: Locking

- When a lock request of a given type is made:
  - If the item is not already locked, the lock is granted
  - If the item is already locked, the lock request must be evaluated to see if it is compatible with the current lock
    - Current: Shared/Read, Requested: Shared/Read – OK
    - Current: Shared/Read, Requested: Exclusive /Write- Nope
    - Current: Exclusive/Write, Requested: Shared/Read– Nope
    - Current: Exclusive/Write, Requested: Exclusive/Write – Nope
  - If a lock is requested but not granted, the requestor must wait doing nothing until the lock is available

Transactions: Managing Concurrency

- A few terms:
  - *Schedule:* A sequence of operations by a set of concurrent transactions that preserves the order of the operations in each individual transaction (no instructions are re-ordered)
  - *Serial schedule:* A schedule where the operations of each transaction are executed consecutively without interleaving (transactions are run one after another)
    - Every serial schedule is guaranteed to not leave the database in an inconsistent state
  - *Non-serial schedule:* A schedule (so no re-ordering w/in a transaction) but where the operations between transactions can be interleaved

Transactions: Schedules

- Our goal is to allow non-serial schedules (so we can have concurrency) which produce the same results as some serial execution.
  - Such a schedule is called *serializable*.

- Some trivial cases:
  - If transactions are only reading, all non-serial schedules are serializable.
  - If two or more transactions are each accessing distinct from each other database tables throughout the transactions, all non-serial schedules are serializable.

Transactions: Schedules and Locking

- The *locking waits* forced by the locking process as described may provide for an interleaved and serializable schedule.
  - If there are significant conflicts between transactions, may turn it into essentially a serial schedule.
- For example, on the next slide locks are a fix to the uncommitted dependency problem:
  - **Uncommitted Dependency:** Occurs when one transaction is allowed to see intermediate results of another transaction before commitment (and the “another” is then aborted)

Transactions: Schedules and Locking

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

Original

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

With Locks
(in this case, the locking essentially
forced a 1-after-other serial execution)

Transactions: Schedules and Locks

- Two issues with locks:
  - Simple locking does not prevent all errors we need to deal with, so we will need to come up with an even stronger approach
  - It is possible to get into deadlock
    - We'll come back to this

Transactions: Schedules and Locks

- Simple locking does not prevent all errors we need to prevent, so we will need to come up with an even stronger approach

- Example:
  - Assume you have two bank accounts, X and Y, with balances 100 and 400 respectively. Two transactions need to be processed:
    - Transfer 100 from account Y to account X
    - Multiply the balances by 1.10 (10% interest!!!)

Transactions: Schedules and Locks

- Initial: X:100 and Y:400
- Transaction T1: Transfer 100 from account Y to account X
- Transaction T2: Multiply the balances by 1.10 (10% interest!!!)

- Two possible serial schedules:
  - T1 first fully, then T2 results in X:220, Y:330
  - T2 first fully, then T1 results in X:210, Y:340
  - Both result in a total of $550.00
- Any concurrent approach needs to result in one of those two outcomes

Transactions: Schedules and Locks

Transaction T1 Makeup

- read(balanceX)
- balanceX = balanceX + 100
- write(balanceX)
- read(balanceY)
- balanceY = balanceY – 100
- write(balanceY)

Transaction T2 Makeup

- read(balanceX)
- balanceX = balanceX * 1.10
- write(balanceX)
- read(balanceY)
- balanceY = balanceY * 1.10
- write(balanceY)

Transactions: Schedules and Locks

The non-interleaved schedule resulting in X:220, Y:330

1. read(balanceX)
2. balanceX = balanceX + 100
3. write(balanceX)
4. read(balanceY)
5. balanceY = balanceY – 100
6. write(balanceY)
7. read(balanceX)
8. balanceX = balanceX * 1.10
9. write(balanceX)
10. read(balanceY)
11. balanceY = balanceY * 1.10
12. write(balanceY)

Transactions: Schedules and Locks

A problematic locking-based interleaved (concurrent) schedule

T1: write_lock(balanceX)
T1: read(balanceX)
T1: balanceX = balanceX + 100
T1: write(balanceX)
T1: release_lock(balanceX)

T2: write_lock(balanceX)
T2: read(balanceX)
T2: balanceX = balanceX *1.1
T2: write(balanceX)
T2: release_lock(balanceX)

T2: write_lock(balanceY)
T2: read(balanceY)
T2: balanceY = balanceY *1.1
T2: write(balanceY)
T2: release_lock(balanceY)
T2: commit

T1: write_lock(balanceY)
T1: read(balanceY)
T1: balanceY = balanceY - 100
T1: write(balanceY)
T1: release_lock(balanceY)
T1: commit

Obeys lock rules!

Resulting values:
X:220
Y:340

Does not match either correct set of balances

Transactions: Schedules and Locking

- 2 Phase Locks (2PL):
  - A transaction follows a 2-phase locking protocol if all locking operations for a transaction precede the first unlock operation in the transaction
  - AKA: Grab all needed locks before releasing any locks (though you don’t have to grab them simultaneously)
- 2PL Theorem: If every transaction in a schedule based off of transactions following the 2PL protocol, that schedule is serializable.

Transactions: Sketch of 2PL Proof

- Here's the idea behind how 2PL Theorem is proved
- Employs *serialization/precedence graph* for a *given schedule*

- Let G = (V,E) be a directed graph as follows
  - Create a node for each transaction

  - Create a directed edge from Ti to Tj if Tj reads the value of an item written previously by Ti (read-write)
  - Create a directed edge from Ti to Tj if Tj writes the value of an item read previously by Ti (write-read)
  - Create a directed edge from Ti to Tj if Tj writes the value of an item written previously by Ti (write-write)

  - The edges represent *conflicting operations*

Transactions: Serialization Graph Example #1

Schedule

T1	T2
1. read(balanceX)	
2. balanceX = balanceX + 100	
3. write(balanceX)	
4. read(balanceY)	
5. balanceY = balanceY – 100	
6. write(balanceY)	
	7. read(balanceX)
	8. balanceX = balanceX * 1.10
	9. write(balanceX)
	10. read(balanceY)
	11. balanceY = balanceY * 1.10
	12. write(balanceY)

T2 reads a value written by T1
T2 writes a value read by T1
T2 writes a value written by T1

all lead to this edge

Transactions: Serialization Graph Example #2

T1: write_lock(balanceX)
T1: read(balanceX)
T1: balanceX = balanceX + 100
T1: write(balanceX)
T1: release_lock(balanceX)

T2: write_lock(balanceX)
T2: read(balanceX)
T2: balanceX = balanceX *1.1
T2: write(balanceX)
T2: release_lock(balanceX)

T2: write_lock(balanceY)
T2: read(balanceY)
T2: balanceY = balanceY *1.1
T2: write(balanceY)
T2: release_lock(balanceY)
T2: commit

T1: write_lock(balanceY)
T1: read(balanceY)
T1: balanceY = balanceY - 100
T1: write(balanceY)
T1: release_lock(balanceY)
T1: commit

T2 reads and writes something (X) after T1

T1 reads and writes something (Y) after T2

Transactions: Sketch of 2PL Proof

- Interleaved execution with a particular schedule is serializable iff the serialization graph of the schedule is acyclic (no cycles)

- If no cycles exist, a topological sort of the nodes in the graph gives the sequential schedules(s) equivalent to the schedule used to build the serialization graph
  - The schedules that order the conflicting operations in the same way

Transactions: Serialization Graph Example #1

Schedule

T1	T2
1. read(balanceX)	
2. balanceX = balanceX + 100	
3. write(balanceX)	
4. read(balanceY)	
5. balanceY = balanceY – 100	
6. write(balanceY)	
	7. read(balanceX)
	8. balanceX = balanceX * 1.10
	9. write(balanceX)
	10. read(balanceY)
	11. balanceY = balanceY * 1.10
	12. write(balanceY)

Serializable schedule
(in fact, this is a strictly serial schedule)

No cycles in graph

T2 reads a value written by T1
T2 writes a value read by T1
T2 writes a value written by T1

all lead to this edge

Transactions: Serialization Graph Example #2

T1: write_lock(balanceX)
T1: read(balanceX)
T1: balanceX = balanceX + 100
T1: write(balanceX)
T1: release_lock(balanceX)

T2: write_lock(balanceX)
T2: read(balanceX)
T2: balanceX = balanceX *1.1
T2: write(balanceX)
T2: release_lock(balanceX)

T2: write_lock(balanceY)
T2: read(balanceY)
T2: balanceY = balanceY *1.1
T2: write(balanceY)
T2: release_lock(balanceY)
T2: commit

T1: write_lock(balanceY)
T1: read(balanceY)
T1: balanceY = balanceY - 100
T1: write(balanceY)
T1: release_lock(balanceY)
T1: commit

T2 reads and writes something (X) after T1

T1 reads and writes something (Y) after T2

We know this is not a serializable schedule (have already shown it doesn't generate same answer as either serial schedule)

Notice cycle in graph as well

Can't map it to T1, T2 serial schedule or T2,T1 serial schedule because neither orders the conflicting operations the same way

Transactions: Sketch of 2PL Proof

- Interleaved execution with a particular schedule is serializable iff the serialization graph of the schedule is acyclic (no cycles)

- Need to show that any schedule 2PL generates results in an acyclic serialization graph

Transactions: Sketch of 2PL Proof

- Assume we have a serialization graph for a 2PL based schedule and there exists an edge Ti→Tj.
  - Means there is a pair of operations Oi from Ti, Oj from Tj which are conflicting (r/w, w/r, or w/w)
  - Since using 2PL (actually, just locking in general even), must have requested locks. The locks would conflict (none of the above lock requests end in both locks granted), so there must be an ordering.
  - Since Tj is the receiving end of the edge, for Oj to execute, Tj must have received the lock, meaning Oi must have executed and Ti must have released its lock.

Transactions: Sketch of 2PL Proof

- Now assume we have a serialization graph for a 2PL based schedule and there exists a path Ti→Tj→Tk
  - Based on locking
    - Ti released a lock before Tj was able to set the lock
    - Tj released a lock (on a possibly different data item) before Tk was able to set the (possibly different data item) lock
  - Based on 2PL, where Tj has to set all locks before releasing any,
    - Ti must have released a lock before Tk was able to set a (possibly different data item) lock
  - Believing in induction, we can extend this to a path of arbitrary length such that Ti → … → Tm means Ti released a lock before Tm set a lock

Transactions: Sketch of 2PL Proof

- Proof by contradiction:
  - Assume there is a cycle in a 2PL-based schedule's serialization graph: Ti→…→Ti.
  - From previous slide, that means Ti released a lock before Ti set a lock.
  - This implies that the transactions are not 2PL, but they are by our original premise of the problem. So, by contradiction, such cycles can't exist in 2PL-based schedule's precedence graphs.
  - Thus, all 2PL-based schedules are serializable.

Transactions: A 2PL-Based Schedule

```
T1: write_lock(balanceX)
T1: read(balanceX)
T1: balanceX = balanceX + 100
T1: write(balanceX)

T2: write_lock(balanceX) // has to wait!

T1: write_lock(balanceY)
T1: read(balanceY)
T1: balanceY = balanceY - 100
T1: write(balanceY)
T1: commit
T1: release_lock(balanceX)
T1: release_lock(balanceY)

// T2 gets the lock now on X
T2: read(balanceX)
T2: balanceX = balanceX *1.1
T2: write(balanceX)

T2: write_lock(balanceY)
T2: read(balanceY)
T2: balanceY = balanceY *1.1
T2: write(balanceY)
T2: commit
T2: release_lock(balanceX)
T2: release_lock(balanceY)
```

This case effectively turns into
T1, T2