Transactions

Kathleen Durant PhD
Northeastern University CS3200
Lesson 9

Outline for the day

- The definition of a transaction
  - Benefits provided
  - What they look like in SQL
- Scheduling Transactions
- Serializability
- Recoverability

What is a transaction?

- A transaction is a collection of operations treated as a single logical operation
  - Typically carried out by a single user or an application program
  - Reads or updates the contents of a database
- A transaction is a ‘logical unit of work’ on a database
  - Each transaction does something in the database
  - No part of it alone achieves anything of use or interest to a user
- Transactions are the unit of recovery, consistency, and integrity of a database
- A *transaction* is the DBMS’s abstract view of a user program: a sequence of reads and writes.

Transactions: ACID Properties

- **Atomicity**: either the entire set of operations happens or none of it does
- **Consistency**: the set of operations taken together should move the system for one consistent state to another consistent state.
- **Isolation**: each system perceives the system as if no other transactions were running concurrently (even though odds are there are other active transactions)
- **Durability**: results of a completed transaction must be permanent - even IF the system crashes

Why the concept of a transaction?

- Real world events require the manipulation of multiple data items
- Examples:
  - A patient's admission to a hospital
  - Transfer of money from checking to savings
  - Adding an additional column to a table
  - Purchase of an item on a website
  - Any other examples?

Example of a transaction

Transfer $50 from account A ($200) to account B ($50)

Read(A);

A -= 50;

Write(A);

Read(B);

B += 50;

Write(B);

Transaction

ACID

- **Atomicity** - shouldn't take money from A without giving it to B
- **Consistency** - money isn't lost or gained
- **Isolation** - other queries shouldn't see A or B change until transaction is completed
- **Durability** - the money does not go back to A is transaction was marked as committed

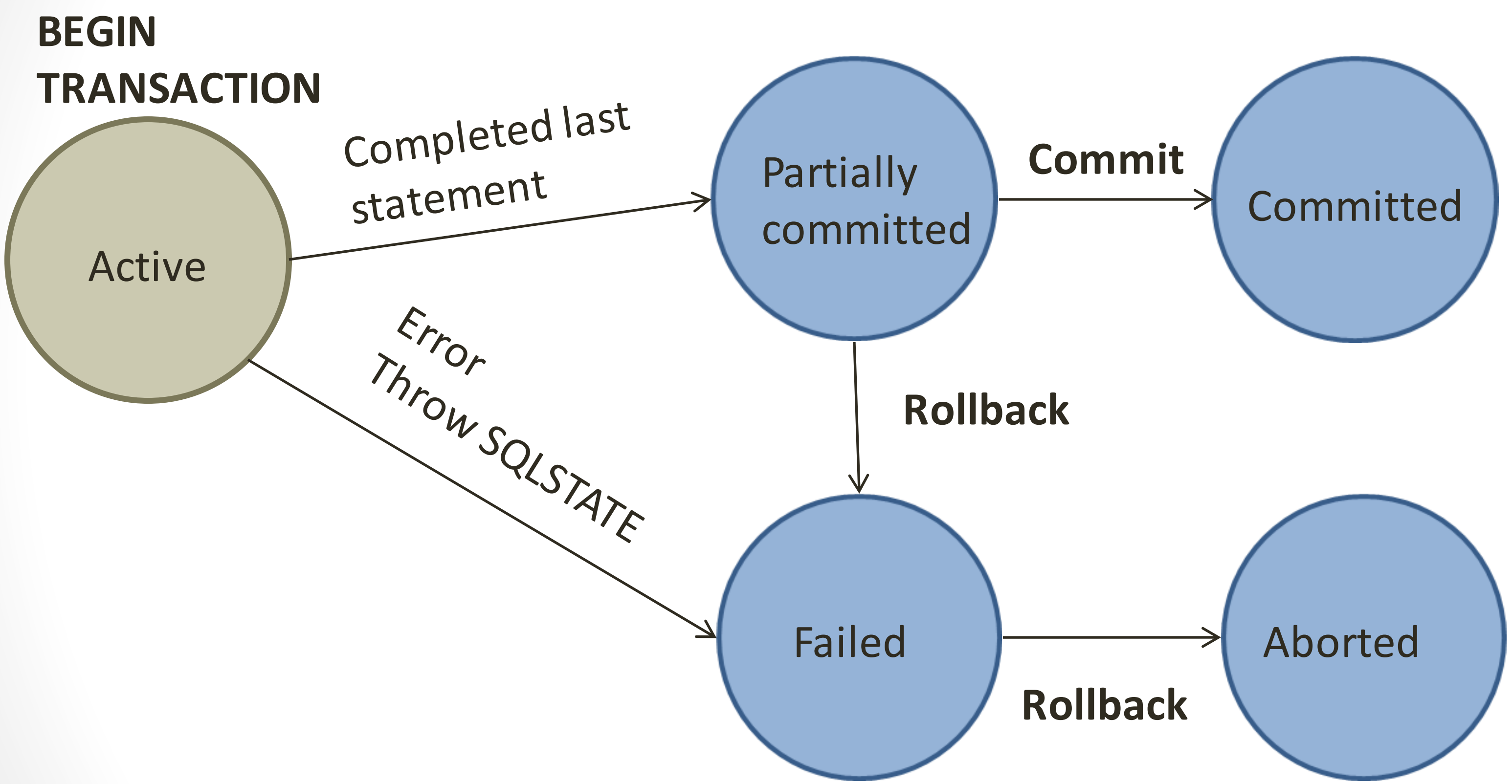
Transaction Commands

- Begin a transaction with the My SQL command:
- **START TRANSACTION**;
- Transactions complete via 2 different input
  - **COMMIT**;
  - **ROLLBACK**;

Transaction Outcomes

- **Can have one of two outcomes:**
  - **Success - transaction *commits* and database reaches a new consistent state.**
  - **Failure - transaction *aborts*, and database must be restored to a consistent state before it started.**
  - **Such a transaction is *rolled back* or *undone*.**
- **Committed transaction cannot be aborted.**
- **An aborted transaction that is rolled back can be restarted later.**

Life of a transaction via a FSM



Simple state machine should help prove the correctness of algorithm

Transaction Abort/Rollback

- Rollback signals the unsuccessful end of a transaction
- Returns the system to the state it was in before the transaction began
- System state must be the same as if the transaction had never existed
- Must abort any transactions that depend on the outcome of the aborting transaction

Transaction Commit

- COMMIT signals the successful end of a Transaction
  - Any changes made by the transaction should be saved
  - These changes are now visible to other transactions
- Declare the transaction permanently complete
- If you commit:
  - No actions should be able to move the DBMS to a state not containing the results of the transaction
  - All operations must be forever persistent in the database

Serial Schedule

- Simplest way to support transaction semantics is to require that each transaction run to completion before the next one begins
- A *schedule* is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions
- A *serial schedule* is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)
- Serial schedule - actual transaction does have the whole database to itself
- This is not a solution for the real world
  - Cannot overlap I/O operations and computation
  - Multiple cores or processors are sitting idle
  - Workload may just be really heavy
  - Response times get long as well as variable
  - Short transactions must wait for long ones to finish

Concurrency Control

Process of managing simultaneous operations on the database without having them interfere with one another.

- **Prevents interference when two or more users are accessing the database simultaneously and at least one is updating data.**
- **Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.**

Serial**izable** schedule: alternative to simple serial schedule

- Multiple transactions running: we know that the execution of a set of simultaneous transactions is correct if it obeys the ACID properties
- More formally:
  - Define the sequence of operations performed is a schedule.
  - Define the sequence of operations performed when running each transaction serially a serial schedule.
  - **Any schedule that *corresponds* to a serial schedule is correct.**

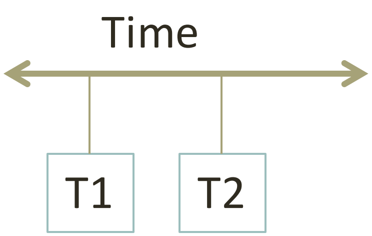
Schedule for a transaction

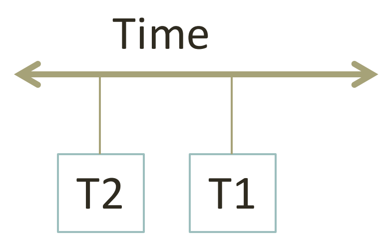
- Actions are reads and writes to the DB
- Transaction: transfer money from account A to account B

Actual Execution	Schedule
Read(A balance)	Read(a)
A balance -=50	
Write (A balance)	Write(a)
Read(B balance)	Read(b)
B_balance +=50;	
Write(B_balance)	Write(b)

Two transactions

- Transfer $50 from Account A to Account B (T1)
- Pay 2% interest to each account (T2)
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
  - **However, the net effect must be equivalent to these two transactions running serially in some order. T2 followed by T1 or T1 followed by t2**





Either Final Balance is Correct

Account A starting balance = $50
Account B starting balance = $200
T1: Transfer $50 from Account A to Account B
T2: Pay 2% interest to each account

T1 then T2

- Apply T1
- Account A = $0
- Account B = $250
- Apply T2
- Account A = $0
- Account B = $255.

T2 then T1

- Apply T2
- Account A = $51
- Account B = $204
- Apply T1
- Account A = $1
- Account B = $254

Examples of Incorrect Schedules

- Increase Balance B by $50
- Increase Balance B by 2%
- Increase Balance A by 2%
- Decrease Balance A by $50

Ending Balances

- Balance B = $255
- Balance A = $1

- Increase Balance B by 2%
- Increase Balance B by $50
- Decrease Balance A by $50
- Increase Balance A by 2%
- Ending Balances
- Balance B = $254.
- Balance A = $0

Account A starting balance = $50
Account B starting balance = $200

Two parallel transactions

- Transfer $50 from Account A to Account B
- Pay 2% interest to each account

Bank pays less interest

Your Transaction	Bank's Transaction	Your Transaction	Bank Transaction
Read(A)		Read(A)	
A balance -=$50			
Write(A balance)		Write(A)	
	Read (A balance)		Read(A)
	A Balance *= 1.02		
	Write(A balance)		Write(A)
	Read(B balance)		Read(B)
	B Balance *= 1.02		
	Write(B Balance)		Write(B)
Read(B Balance)		Read(B)	
B Balance +=$50			
Write(B Balance)		Write(B)	

Serializability

- **Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.**
- **Could run transactions serially, but this limits degree of concurrency or parallelism in the system.**
- **Serializability identifies those executions of transactions guaranteed to ensure consistency.**

21

Serializability

Schedule

Sequence of reads/writes by set of concurrent transactions.

Serial Schedule

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- **No guarantee that results of all serial executions of a given set of transactions will be identical.**

Nonserial Schedule

- **Schedule where operations from set of concurrent transactions are interleaved.**
- **Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.**
- **In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called *serializable*.**

Serializability

- **In serializability, ordering of read/writes is important:**

  (a) If two transactions only read a data item, they do not conflict and order is not important.

  (b) If two transactions either read or write separate data items, they do not conflict and order is not important.

  (c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Lost Update Problem

- **Successfully completed update is overridden by another user.**
- **T_1 withdrawing \$10 from an account with bal_x, initially \$100.**
- **T_2 depositing \$100 into same account.**
- **Serially, final balance would be \$190.**

25

Lost Update Problem

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

- **Loss of T_2's update avoided by preventing T_1 from reading bal_x until after update.**

26

Uncommitted Dependency Problem

- Occurs when one transaction can see intermediate results of another transaction before it has committed.
- T_4 updates bal_x to $200 but it aborts, so bal_x should be back at original value of $100.
- T_3 has read new value of bal_x ($200) and uses value as basis of $10 reduction, giving a new balance of $190, instead of $90.

Uncommitted Dependency Problem

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

- **Problem avoided by preventing T_3 from reading bal_x until after T_4 commits or aborts.**

28

Inconsistent Analysis Problem

- **Occurs when transaction reads several values but second transaction updates some of them during execution of first.**
- **Sometimes referred to as *dirty read* or *unrepeatable read*.**
- **T_6 is totaling balances of account x ($100), account y ($50), and account z ($25).**
- **Meantime, T_5 has transferred $10 from bal_x to bal_z, so T_6 now has wrong result ($10 too high).**

Inconsistent Analysis Problem

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

- **Problem avoided by preventing T_6 from reading bal_x and bal_z until after T_5 completed updates.**

Recoverability

- Since we need to fix things up after a failed transaction in addition to serializability we also need recoverability
- If transaction Tj depends on transaction Ti and Ti aborts, then Tj must also abort
- **Thus our goal is to find schedules that are both serializable and recoverable**

An Unrecoverable Schedule

Schedule Unrecoverable		
	T1	T2
20	Read (A)	
30	Write (A)	
30		Read (A)
40		Write (A)
	Read (B)	Commit
	Rollback	

T2 Commits

T1 Aborts

- What do we have to do when T1 aborts?
- Undo both T1 & T2.
- Since T2 is already committed this is called an Unrecoverable schedule

Aborting transactions

- All actions of aborted transactions have to be undone
  - Dirty read can result in unrecoverable schedule
  - T1 writes A, then T2 reads A and makes modifications based on A's value
  - T2 commits, and later T1 is aborted
- T2 worked with invalid data and hence has to be aborted as well; but T2 already committed…
  - Recoverable schedule: cannot allow T2 to commit until T1 has committed
  - Can still lead to cascading aborts

Aborting transactions

- Data produced by an uncommitted transaction is called dirty
- If a transaction produced dirty data and then aborts then all transactions that read the dirty data must also abort
- Such abort dependencies are called cascading aborts and should be avoided
- Why?

Performance impact
Complicated to maintain dependency relationships
What if events are user visible?

Preventing anomalies through locking

- DBMS can support concurrent transactions while preventing anomalies by using a locking protocol
  - If a transaction wants to read an object, it first requests a shared lock (S-lock) on the object
  - If a transaction wants to modify an object, it first requests an exclusive lock (X-lock) on the object
  - If requested lock is not available – then the transaction waits
- Multiple transactions can hold a shared lock on an object
- At most one transaction can hold an exclusive lock on an object

Lock-based Concurrency control

- Strict Two-phase Locking (Strict 2PL) Protocol:
  - Each transaction must obtain the appropriate lock before accessing an object.
  - All locks held by a transaction are released when the transaction is completed.
- All this happens automatically inside the DBMS
  - Strict 2PL allows only serializable schedules.
  - Prevents all the anomalies shown earlier
- Two phases in lock algorithm
  - Growing phase where locks are acquired on resources
  - Shrinking phase where locks are released

Phantom Problem

- Assume initially the youngest sailor is 20 years old
  - T1 contains this query twice
  - SELECT rating, MIN(age) FROM Sailors (Query Q)
  - T2 inserts a new sailor with age 18
- Consider the following schedule:
- T1 runs query Q, T2 inserts new sailor, T1 runs query Q again
  - T1 sees two different results! Unrepeatable read.
  - Would Strict 2PL prevent this?
- Assume T1 acquires Shared lock on each existing sailor tuple
  - T2 inserts a new tuple, which is not locked by T1
  - T2 releases its Exclusive lock on the new sailor before T1 reads Sailors again
- What went wrong?

Lock level of objects

- T1 cannot lock a tuple that T2 will insert
  - …but T1 could lock the entire Sailors table
- Now T2 cannot insert anything until T1 completed
- What if T1 computed a slightly different query:
  - SELECT MIN(age) FROM Sailors WHERE rating = 8
  - Now locking the entire Sailors table seems excessive, because inserting a new sailor with rating <> 8 would not create a problem
  - T1 can lock the predicate [rating = 8] on Sailors
- General challenge: DBMS needs to choose appropriate granularity for locking

Deadlocks

- Assume T1 and T2 both want to read and write objects A and B
  - T1 acquires X-lock on A;
  - T2 acquires X-lock on B
- Now T1 wants to update B, but has to wait for T2 to release its lock on B
- But T2 wants to read A and also waits for T1 to release its lock on A
- Strict 2PL does not allow either to release its locks before the transaction completed. Deadlock!
- DBMS can detect this
- Automatically breaks deadlock by aborting one of the involved transactions
  - Tricky to choose which one to abort: work performed is lost

Performance of Locking

- Locks force transactions to wait
  - Abort and restart due to deadlock wastes the work done by the aborted transaction
  - In practice, deadlocks are rare, e.g., due to lock downgrades approach
- Waiting for locks becomes bigger problem as more transactions execute concurrently
  - Allowing more concurrent transactions initially increases throughput, but at some point leads to thrashing
  - Need to limit maximum number of concurrent transactions to prevent thrashing
  - Minimize lock contention by reducing the time a transaction holds locks and by avoiding hotspots (objects frequently accessed)

Controlling Locking Overhead

- Declaring transaction as "READ ONLY" increases concurrency
- Isolation level: trade off concurrency against exposure of transaction to other transaction's uncommitted changes
  - Degrees of serializability

Isolation level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Locking versus Isolation level

- SERIALIZABLE: obtains locks on (sets of) accessed objects and holds them until the end
- REPEATABLE READ: same locks as for serializable transaction, but does not lock sets of objects at higher level
- READ COMMITTED: obtains X-locks before writing and holds them until the end; obtains S-locks before reading, but releases them immediately after reading
- READ UNCOMMITTED: does not obtain S-locks for reading; not allowed to perform any writes
  - Does not request any locks ever

My SQL and transactions

- A transaction is implicitly created every time you issue a SQL command (called AUTOCOMMIT)
- SQL transactions must be serializable and recoverable, where serializable is defined as having the same effect as a serial execution
- Commit transaction with **commit;**
- Abort transaction with **rollback;**

Precedence Graph

- To determine if a schedule is conflict serializable we use a precedence graph
- Transactions are vertices of the graph
- There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serial schedule
- Edge T1 -> T2 if in the schedule we have:
  - T1 Read(R) followed by T2 Write(R) for the same resource R
  - T1 Write(R) followed by T2 Read(R)
  - T1 Write(R) followed by T2 Write(R)
- The schedule is serializable if there are no cycles

Are these schedules view equivalent?

Schedule U	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)
Read (B)	
Write (B)	

Schedule T	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

- No – In Schedule U T2 reads initial value of B
- While in Schedule T T1 reads initial value of B

Summary

- Concurrency control is one of the most important functions provided by a DBMS.
  - Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and can schedule actions of different transactions in such a way as to ensure that the resulting execution is equivalent to executing the transactions one after the other in some order.
- DBMS automatically undoes the actions of aborted transactions.
- Consistent state: Only the effects of committed transactions seen.