Transaction Management: Introduction (Chap. 16)

CS634
Class 14

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

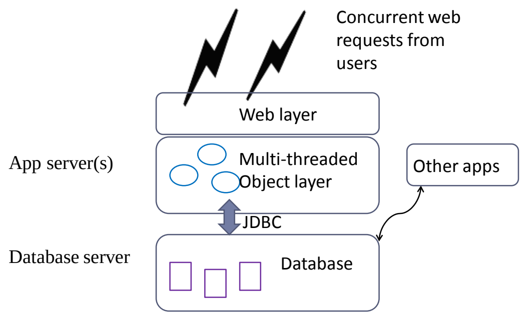
What are Transactions?

- So far, we looked at individual queries; in practice, a task consists of a *sequence* of actions
- E.g., "Transfer $1000 from account A to account B"
  - Subtract $1000 from account A
  - Subtract transfer fee from account A
  - Credit $1000 to account B
- A *transaction* is the DBMS's view of a user program:
  - Must be interpreted as "unit of work": either entire transaction executes, or no part of it executes/has any effect on DBMS
  - Two special final actions: COMMIT or ABORT

Concurrent Execution

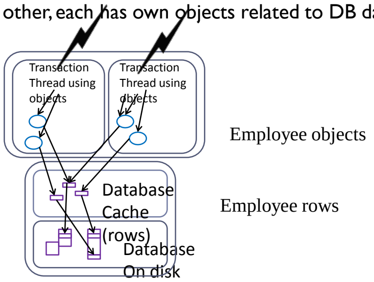
- DBMS receives large numbers of concurrent requests
  - Concurrent (or parallel) execution improves performance
  - Two transactions are *concurrent* if they overlap in time.
  - Disk accesses are frequent, and relatively slow; CPU can do a lot of work while waiting for the disk, or even SSD
  - Goal is to increase/maximize system throughput
    - Number of transactions executed per time unit
- Concurrency control
  - Protocols that ensure things execute correctly in parallel
  - Broad and difficult challenge that goes beyond DBMS realm
    - OS, Distributed Programming, hardware scheduling (CPU registers), etc
  - Our focus is DBMS, but some principles span beyond DBMS

Major Example: the web app



Web app in execution (CS636)

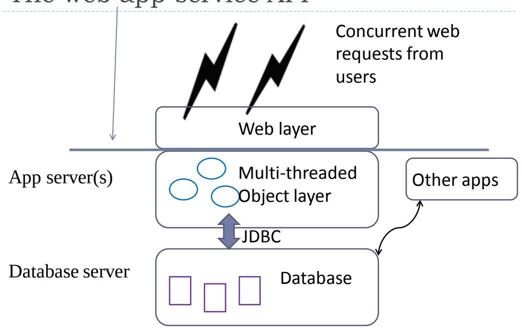
- To keep transactions executing concurrently, yet isolated from each other, each has own objects related to DB data



Web app Transactions

- Each application action turns into a database transaction
- A well-designed app has a "service API" describing those actions
- A request execution calls the service API one or more times.
- Each service call represents an application action and contains a transaction
- Thus transactions are contained in request-response cycles
- This ensures that transactions are short-lived, good for performance
- But they still can run concurrently under high-enough load

The web app service API



ACID Properties

Transaction Management must fulfill four requirements:

1. Atomicity: either all actions within a transaction are carried out, or none is
   - Only actions of committed transactions must be visible
2. Consistency: concurrent execution must leave DBMS in consistent state
3. Isolation: each transaction is protected from effects of other concurrent transactions
   - Net effect is that of **some sequential execution**
4. Durability: once a transaction commits, DBMS changes will persist
   - Conversely, if a transaction aborts/is aborted, there are no effects

Roles of Transaction Manager

- Concurrency Control
  - Ensuring correct execution in the presence of multiple transactions running in parallel
- Crash recovery
  - Ensure that atomicity is preserved if the system crashes while one or more transactions are still incomplete
  - Main idea is to keep a log of operations; every action is logged before execution (Write-Ahead Log or WAL)

Modeling Transactions

- User programs may carry out many operations …
  - Data-related computations
  - Prompting user for input, handling web requests
- … but the DBMS is only concerned about what data is read/written from/to the database
- A *transaction* is abstracted by a sequence of time-ordered read and write actions
  - e.g., R(X), R(Y), W(X), W(Y)
  - R=read, W=write, data element in parentheses
  - Each individual action is indivisible, or atomic
  - SQL UPDATE = R(X) W(X)

Important dataflow assumptions

- Transactions interact with one another as they run only via database read and write operations.
  - No messages exchanged between transactions
  - No use of shared memory between transactions
  - Oracle, other DBs, enforce this
- Transactions may accept information from the environment when they start and return information to the environment when they finish by committing.
  - The agent that starts a transaction will come to know whether it committed or aborted, and can act on that information.
  - Thus it is possible for data to go from one transaction to the environment and then to another starting transaction, but note that these transactions are not concurrent.

Scheduling Transactions

- *Serial schedule:* no interleaving of transactions
  - Safe, but poor performance!
- *Schedule equivalence:* two schedules are equivalent if they lead to the same state of the DBMS (see footnote on pg. 525 that includes values returned to user in relevant "state")
- *Serializable schedule:* schedule that is equivalent to some serial execution of transactions
  - But still allows interleaving/concurrency!

Serializable schedule example

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

- Same effect as executing T1 completely, then T2

If execution is not serializable…

- Non-serializable concurrent executions can show anomalies, i.e., clearly bad behavior
- Let's look at some examples

Concurrency: lost update anomaly

- Consider two transactions (in a really bad DB) where A = 100

T1:	A = A + 100
T2:	A = A + 100

- T1 & T2 are concurrent, running same transaction program
- T1& T2 both read old value, 100, add 100, store 200
- One of the updates has been lost!
- Consistency requirement: after execution, A should reflect all deposits (Money should not be created or destroyed)
- No guarantee that T1 will execute before T2 or vice-versa…
- … but the net effect must be equivalent to these two transactions running one-after-the-other in ***some*** order

Concurrency: more complex case (1/3)

- Consider two transactions running different programs

T1:	A=A+100,	B=B-100
T2:	A=1.06*A,	B=1.06*B

- T1 performs an account transfer
- T2 performs credit of (6%) interest amount
- Consistency requirement: after execution, sum of accounts must be 106% the initial sum (before execution)
- No guarantee that T1 will execute before T2 or vice-versa…
- … but the net effect must be equivalent to these two transactions running one-after-the-other in ***some*** order

Concurrency: when things go wrong (2/3)

- Assume that initially there are $500 in both accounts
- Consider a possible *interleaving* or *schedule*

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

- After execution, A=636, B=424, A+B=1060

CORRECT

Concurrency: when things go wrong (3/3)

- Consider another interleaving or schedule:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- After execution, A=636, B=430, A+B=1066

WRONG!!!

- The DBMS view

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

Concurrent Execution Anomalies

- Anomalies may occur in concurrent execution
- The notion of conflicts helps understand anomalies
- Is there a conflict when multiple READ operations are posted? No
- What if one of the operations is a WRITE? YES!
- WR, RW and WW conflicts

WR Conflicts

- Reading Uncommitted Data (Dirty Reads)

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

- The earlier example where interest is not properly credited is due to a WR conflict
- Value of *A* written by *T1* is read by *T2* before *T1* completed all its changes

RW Conflicts

- Unrepeatable Reads

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

- Scenario: Let *A (=1)* be the number of copies of an item. *T1* checks the number available. If the number is greater than *0*, *T1* places an order by decrementing the count
- In the meantime, *T2* updated the value of the count (say, to zero)
- *T1* will set the count to a negative value!

WW Conflicts

- Overwriting Uncommitted Data

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

- Assume two employees must always have same salary
- *T1* sets the salaries to $1000, *T2* to $2000
- There is a "lost update", and the final salaries are $1000 and $2000
- "Lost" update because the transaction that comes last in serial order should set both values. One got lost.

Scheduling Transactions: recall terminology

- *Serial schedule:* no interleaving of transactions
  - Safe, but poor performance!
- *Schedule equivalence:* two schedules are equivalent if they lead to the same state of the DBMS (see footnote on pg. 525 that includes values returned to user in relevant "state")
- *Serializable schedule:* schedule that is equivalent to some serial execution of transactions
  - But still allows interleaving/concurrency!

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule *S* is conflict serializable if *S* is conflict equivalent to some serial schedule
- A conflict serializable schedule is serializable (to be shown in future classes)
- Some other schedules are also serializable

Why is serializability important?

- If each transaction preserves consistency, every serializable schedule preserves consistency
  - For example, transactions that move money around should always preserve the total amount of money.
  - If running with serializable transactions, we only need to check that each transaction program has this property, and we know that the system does.
- How to ensure serializable schedules?
  - Use **locking** protocols (ensuring conflict serializability)
  - DBMS inserts proper locking actions, user is oblivious to locking (except through its effect on performance, and deadlocks)
  - There are other ways too, covered later.

Strict Two-Phase Locking (Strict 2PL)

- Protocol steps
  - Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held are released when the transaction completes
    - (Non-strict) 2PL: Release locks anytime, but cannot acquire locks after releasing any lock.
- Strict 2PL allows only serializable schedules.
  - It simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Strict 2PL Example (red op is blocked)

T1: S(A) R(A) S(B)
T2: S(A) R(A) X(B)

T1: S(A) R(A) S(B)
T2: S(A) R(A) X(B) R(B)

T1: S(A) R(A) S(B)
T2: S(A) R(A) X(B) R(B) W(B) C

T1: S(A) R(A) S(B) R(B) C
T2: S(A) R(A) X(B) R(B) W(B) C

Aborting Transactions

- When *Ti* is aborted, all its actions have to be undone
  - if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
  - *cascading aborts* can be avoided with 2PL by releasing locks only at commit (Strict 2PL)
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits
  - This also means the schedule is "recoverable": transactions commit only after all transactions whose changes they read commit.
  - In general, recoverable and serializable are separate properties of concurrency protocols, but Strict 2PL has both.
- Strict 2PL is recoverable, and cascading aborts are prevented
  - At the cost of decreased concurrency
  - No free lunch!
  - Increased parallelism leads to locking protocol complexity

Deadlocks

- Cycle of transactions waiting for locks to be released by each other

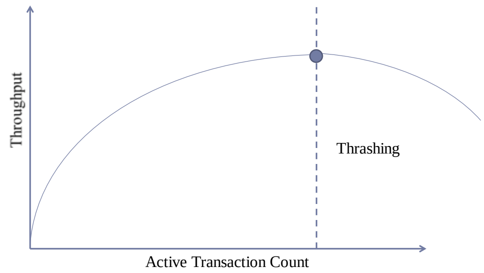
T1: X(A) W(A) S(B) [R(B) …]
T2: X(B) W(B) S(A) [R(A) …]

- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Locking Performance

- Lock-based schemes rely on two mechanisms
  - Blocking
  - Aborting
- Both blocking and aborting cause performance overhead
  - Transactions may have to wait
  - Transactions may need to be re-executed
- How does blocking affect throughput?
  - First few transactions do not conflict – no blocking
    - Parallel execution, performance increase
  - As more transactions execute, blocking occurs
  - After a point, adding more transactions decreases throughput!

Locking Performance (2)



Improving Performance

- Locking the smallest-sized objects possible
  - e.g., row set instead of table
- Reduce the time a lock is held for
  - Release locks faster
- Reducing hot spots
  - Careful review of application design
  - Reduce contention

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations

Transaction Support in SQL

- A transaction is automatically started when user executes a statement or accesses the catalogs
- Transaction is either committed (COMMIT) or aborted (ROLLBACK)
- New in SQL-99: SAVEPOINT feature

  SAVEPOINT <savepoint name>
  Actions …
  ROLLBACK TO SAVEPOINT <savepoint name>
- SAVEPOINT advantage vs. sequence of transactions
  - Can roll back over multiple savepoints
  - Lower overhead: no new transaction initiated (book, pg. 536)
  - But transaction initiation is not an expensive action. Locks are still held on changes done before savepoint, when rollback to savepoint done. Locks would be released if a real commit is done.
  - Conceivably of use for "what-if" calculations, but hard to find examples.

Setting Transaction Properties in SQL

- Access Mode
  - READ ONLY vs READ WRITE
- Isolation Level (decreasing level of concurrency)

Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	No	Possible	Possible
REPEATABLE READ	No	No	Possible
SERIALIZABLE	No	No	No

- We haven't yet seen an example of a phantom—next time.

Isolation Levels in Practice

- Databases default to RC, read-committed, so many apps run that way, can have their read data changed, and phantoms
- Web apps (JEE, anyway) have a hard time overriding RC, so most are running at RC
- The 2PL locking scheme we studied was for RR, repeatable read: transaction takes long term read and write locks
- Long term = until commit of that transaction

Read Committed (RC) Isolation

- 2PL can be modified for RC: take long-term write locks but not long term read locks
- Reads are atomic as operations, but that's it
- Lost updates can happen in RC: system takes 2PC locks only for the write operations:
  R1(A)R2(A)W2(B)C2W1(B)C1
  R1(A)R2(A)X2(B)W2(B)C2X1(B)W1(B)C1 (RC isolation)
- Update statements are atomic, so that case of read-then-write is safe even at RC
- Update T set A = A + 100 (safe at RC isolation)
- Remember to use update when possible!

Syntax for SQL

SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE READ WRITE

SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ READ ONLY

- Note:
  - READ UNCOMITTED cannot be READ WRITE