Relational Database System Implementation

CS122 – Lecture 19

Winter Term, 2018-2019

Last Time: Two-Phase Locking

- Require that transactions manage locks in two phases
- *Growing* phase:
  - A txn may acquire new locks, and may not release any lock
- *Shrinking* phase:
  - A txn may release locks, and may not acquire any new locks
- Transactions start in the growing phase
  - As transaction operates on various data items, it acquires locks on those items
- Once a txn releases any lock, it enters the shrinking phase
  - It can only release locks, until all of its locks are released
- Two-phase locking protocol only allows conflict-serializable execution schedules

Two-Phase Locking Example

- Previous example, updated to follow two-phase rule:
  - Now we know it is conflict-serializable
- What <u>new</u> problem do we have?
  - Shared and exclusive locks are incompatible…
- A schedule executing these transactions is prone to deadlock!

T_i:

```
lock-X(B);
read(B);
$B := B - 30$;
write(B);
lock-X(A);
read(A);
$A := A + 30$;
write(A);
unlock(B);
unlock(A);
commit.
```

T_j:

```
lock-S(A);
read(A);
lock-S(B);
read(B);
unlock(A);
unlock(B);
display($A + B$);
commit.
```

2PL and Deadlocks

- A two-phase locking schedule that deadlocks:
- Can't avoid this issue…
  - Never know what data items a transaction might use!
- Only recourse is to identify deadlocks when they occur
  - Choose one transaction in the deadlock, and abort it.
  - Aborted transaction is called the *victim*

T_i:	T_j:
lock-X(B);	
read(B);	
$B := B - 30$;	
write(B);	
	lock-S(A);
	read(A);
lock-X(A); WAIT	
read(A);	lock-S(B); WAIT
$A := A + 30$;	read(B);
write(A);	unlock(A);
unlock(B);	unlock(B);
unlock(A);	display($A + B$);
commit.	commit.

2PL: Detecting Deadlocks

- Current Lock Manager design:
  - Lock manager tracks every data item that is locked
    - Lock manager records the transaction that has the item locked, and the lock mode (shared or exclusive)
  - If other transactions are waiting to lock a data item, the lock manager also records these lock-requests
- The lock manager also maintains a *waits-for graph*, tracking relationships between waiting transactions
  - If a transaction T_i holds a lock on a data item Q, and T_j is waiting to lock Q, the waits-for graph records $T_j \rightarrow T_i$

2PL: Detecting Deadlocks (2)

- If waits-for graph contains a cycle, a deadlock exists!
  - <u>All</u> transactions in the cycle are deadlocked, not just one
- How many outgoing edges will a transaction have in the waits-for graph?
  - Depends on the mode of the current lock on the item!
  - e.g. if item is locked in shared-mode by multiple txns, and an exclusive-mode request is made, requester will have outgoing edges to all txns holding the lock
- Multiple deadlock cycles could exist in waits-for graph
  - One transaction could be involved in multiple cycles
  - Deadlock detection must identify <u>all</u> cycles in graph

2PL: Detecting Deadlocks (3)

- Waits-for graph can be updated every time a request cannot be granted immediately
  - If a request can be granted immediately, no reason to update the waits-for graph… transaction isn't waiting…
- When a transaction unlocks a data item, one or more waiting requests can be granted
  - Must again update the waits-for graph
- When a txn aborts, all of its locks and outstanding requests are removed from the lock manager
  - Again, must update the waits-for graph

2PL: Detecting Deadlocks (4)

- When should deadlock detection be invoked?
  - Will certainly consume CPU resources, so don't want to run it all the time
- Don't need to run it all the time…
  - Deadlocks have a nice property: they don't go away!
- Only need to consider running deadlock detection when a lock request can't be granted right away
  - e.g. if a lock request isn't satisfied within a specific time interval, invoke deadlock detection algorithm

2PL: Resolving Deadlocks

- If deadlock is detected, another important question:
- How should we choose a victim transaction to abort?
- Example:
  - Transaction T_1 is performing a long-running analysis
  - Transaction T_2 involves three quick operations
  - If T_1 and T_2 deadlock, which should be aborted?
  - Preferably, T_2 should be aborted so that less work is lost
- Goal:
  - Choose a victim to abort that will incur the least cost

2PL: Resolving Deadlocks (2)

- Identifying victim that will incur least cost is difficult to do
- Can consider definite measures:
  - How long each transaction in the deadlock cycle has been running
    - Abort the youngest transaction in the cycle?
  - How costly the transaction itself will be to abort:
    - How many data-items has the transaction modified?
    - The more writes the transaction has performed, the more costly it will be to rollback all changes
  - How many deadlock cycles the transaction is involved in
    - <u>Every</u> deadlock cycle must be broken! If multiple cycles can be broken by aborting one transaction, everybody [else] wins.

2PL: Resolving Deadlocks (3)

- Can also try to predict the future:
  - How close is each transaction to being finished?
    - If not throwing away a large amount of work, would be nice to abort transactions that still have a long way to go
  - How many more data items will the transaction need?
    - Prefer to abort a transaction that requires more resources over one that requires less
  - Can be challenging to make these predictions, but the set of queries against a DB usually doesn’t vary a lot

- Or, just pick one randomly ☺

Two-Phase Locking Protocol

- So far, two-phase locking protocol ensures conflict-serializable execution schedules…
- …but we really wanted strict schedules.
  - Rules out cascading aborts, nonrecoverable schedules, and complicated recovery processing

- The form of 2PL previously introduced is called basic two-phase locking

Basic Two-Phase Locking

- Want to modify 2PL protocol to only allow strict transaction schedules
- A schedule *S* is *strict* if, for every pair of txns T_i and T_j:
  - If T_j reads or writes a data-item previously written by T_i, then T_j is not allowed to do this until T_i first commits

- What could T_i do to ensure that T_j can't read or write a data item T_i has written to, until T_i commits?
- Simple: hang onto its write-locks until after it commits!

Strict Two-Phase Locking

- *Strict two-phase locking* extends basic 2PL:
  - Transactions must still follow the two phases of 2PL
  - A transaction must also hold on to all exclusive locks until after the transaction commits
- Ensures the strict schedule requirement is satisfied
  - Prevents all undesirable behaviors we want to eliminate

- Strict 2PL is sufficient to build a standalone database, but most commercial DBs don't actually use it!

Distributed Databases

- *Distributed databases* are increasingly necessary for handling massive numbers of clients
  - Very large companies (banks, credit companies, etc.); huge number of clients distributed around the world
- Either undesirable or infeasible to handle all database transactions with a single system
  - Size of data-set may be far too large for a single server
  - Want to be aware of network topology – clients should interact with servers topologically “close” to them
  - Also reduces risk of service outages for clients, if either a specific server fails, or connectivity between servers fails

Distributed Databases (2)

- *Homogeneous* distributed databases
  - All servers use the same DBMS software, and generally collaborate very closely (e.g. same schemas, queries)
- *Heterogeneous* distributed databases
  - Different servers may use different DBMS software, etc.
- Still want to make transaction-processing guarantees for such systems…
  - Each database has its own concurrency control system
  - Different databases may even use different concurrency-control mechanisms

Global Serializability

- Individual sites participate in *distributed transactions* with each other
    - A transaction that spans multiple database systems
- Individual databases can ensure that local transactions are executed according to some serializable schedule…
- Doesn’t automatically guarantee the *global* transaction schedule is also serializable!
- Want distributed DB to enforce *global serializability*
- To do this, strict two-phase locking is not enough

Rigorous Two-Phase Locking

- *Rigorous two-phase locking* is a further modification of strict two-phase locking:
  - Transactions follow the two phases of 2PL…
  - A transaction must hold on to <u>all</u> locks (shared *or* exclusive) until after the transaction commits
- Also known as *strong-strict two-phase locking*
- Has a useful property for distributed databases:
  - Transactions can be serialized in the order they commit
  - Allows for efficient and scalable distributed transaction processing that satisfies the ACID properties

Rigorous Two-Phase Locking (2)

- Commercial databases with lock-based concurrency control actually use rigorous 2PL, not strict 2PL
  - Allows them to be used in distributed database systems
- Does rigorous 2PL actually have two phases?
  - Transactions follow the two phases of 2PL…
  - …and a transaction must hold on to <u>all</u> locks (shared *or* exclusive) until after the transaction commits
- Second phase is always performed entirely in one shot, after commit. Not really a "phase"…

Conditional Operations

- Transactions don't always know what data-items they will need to lock, or what locks they will need to use
- Example: conditional operation
  - If A >= 100 then A := A – 100, otherwise B := B – 100.
  - Transaction reads A first, then either modifies A or B.
- What lock should we initially acquire against A?
  - Depends on current value of A…
  - Could simply play it safe, and always lock-X(A)
  - But, this would reduce the concurrency of the system

Lock Conversions

- A better approach:
  - Allow txns to *upgrade* a shared lock into an exclusive lock
  - Similarly, allow them to *downgrade* an exclusive lock into a shared lock
- As before, lock conversions must follow 2PL protocol:
  - Lock upgrades are only allowed during growing phase
  - Lock downgrades are only allowed during shrinking phase
- Two new operations to support with Lock Manager:

upgrade(Q)	Upgrades a shared lock on Q to exclusive
downgrade(Q)	Downgrades exclusive lock on Q to shared

Lock Conversions (2)

- Gives us two options with previous example
  - If A >= 100 then A := A – 100, otherwise B := B – 100.
- Option 1:
  - Acquire a shared lock on A
  - If A >= 100 then upgrade to exclusive lock on A
  - Otherwise, acquire an exclusive lock on B (must retain shared lock on A)

```
T_i:  lock-S(A);
      read(A);
      Hey, A > 100!
      upgrade(A);
      A := A – 100;
      write(A);
      commit;
      unlock(A).
```

Lock Conversions (3)

- Gives us two options with previous example
  - If A >= 100 then $A := A - 100$, otherwise $B := B - 100$.
- Option 2:
  - Acquire an exclusive lock on A
  - If $A < 100$ then downgrade to shared lock on A and acquire an exclusive lock on B…
- *Actually, not allowed to do it in that order!!!*
  - Downgrading releases a lock; causes transaction to enter the shrinking phase!
  - Not allowed to acquire another lock on B

Lock Conversion Risks

- What happens with this transaction schedule?
  - T_i and T_j both acquire shared locks on A
  - T_i tries to upgrade its lock, but is blocked by T_j
  - T_j tries to upgrade its lock, but is blocked by T_i
- T_i and T_j end up deadlocked
  - One of them must be aborted by the database

T_i:	T_j:
lock-S(A); read(A);	
	lock-S(A); read(A);

upgrade(A);
$A := A - 100$;
write(A);	

	upgrade(A);
	$A := A - 100$;
	write(A);

commit; unlock(A).	
	commit; unlock(A).

Lock Conversion Risks (2)

- What if both T_i and T_j tried to acquire exclusive locks right away?
  - Reduces opportunities for concurrency, but avoids deadlocks!
- One reason why databases often provide modifiers to SELECT statements, such as:

  SELECT … FOR UPDATE ;

  - Acquires exclusive locks, not shared locks

T_i:	T_j:
lock-X(A);	
read(A);	
A := A – 100;	
write(A);	
commit;	
unlock(A).	
	lock-X(A);
	read(A);
	A := A – 100;
	write(A);
	commit;
	unlock(A).

Typical Lock-Conversion Strategy

- Most databases use rigorous 2PL with lock-upgrading
- When a transaction T_i issues a read(Q) operation:
  - If DB *knows* that the SQL command is going to read and then write Q, it can issue lock-X(Q) first, then read(Q)
    - e.g. "UPDATE t SET a = a + 5" must read and write data values
  - Otherwise, DB issues a lock-S(Q) first, then read(Q)
- When T_i issues a write(Q):
  - If T_i already has a shared lock on Q then DB issues upgrade(Q), then write(Q)
  - Otherwise, DB issues a lock-X(Q) first, then write(Q)
- All locks are released after transaction T_i commits