Concurrency Control

- The main challenge with concurrency and transactions is to preserve *isolation* — each transaction operates as if it were the only one running on a database
- We've seen how isolation can be preserved through *serializability* — scheduling the operations in a transaction so that the schedule is equivalent to one where the transactions are executed in sequence
- The concepts here assume that transactions are relatively brief; advanced approaches dealing with long-duration transactions are beyond this scope

Locks

- *Locks* are a time-honored mechanism for enforcing *mutual exclusion* — that is, allowing at most one transaction to access a data item at any given moment
- The mutual exclusion results in the desired isolation
- Two main lock modes:
  - *Shared-mode lock*: a transaction can read, but not write, some data item *Q* (denoted by *S*)
  - *Exclusive-mode lock*: a transaction can both read and write item *Q* (denoted by *X*)

Basic Lock Sequence

At its most basic level, the locking sequence follows these steps for some transaction *T*:

1. Transaction *T* requests a lock for data item *Q* (written as *lock-S(Q)* for shared locks and *lock-X(Q)* for exclusive locks)
2. Concurrency-control manager *grants* the lock request if the requested lock is *compatible* with currently active locks; otherwise *T* has to wait
3. When lock is granted, *T* proceeds, calling *unlock(Q)* when it no longer needs the lock

Lock Compatibility

- Very straightforward — only shared locks on a specific data item *Q* are compatible, as seen below
- The upshot: we can have any number of shared locks on *Q*, but once an exclusive lock is requested, *all* shared locks have to be released first, then no other locks can be granted until the exclusive lock is released

comp	S	X
S	true	false
X	false	false

Locking Issues

- Unlocking Q right after a transaction T_i is done with it risks *inconsistency* — other transactions T_j may see an inconsistent big picture before T_i is totally done
- On the other hand, if T_i locks Q for too long, it will force another transaction T_j to wait for *unlock(Q)* — but if T_j currently locks another item R that T_i needs later, then T_i is forced to wait too, resulting in *deadlock*
- Finally, a transaction may be *starved* if it needs an exclusive lock on Q but new transactions keep getting a shared lock on Q

Locking Protocols

- We prefer deadlock over inconsistency: we can roll back upon deadlock, but inconsistency leads to errors
- Starvation isn't too hard to avoid — just give priority to older lock requests before granting new ones
- *Locking protocols* state how locks are managed
- Given a locking protocol L: T_i *precedes* T_j if T_j needs a lock for which T_i already holds an incompatible lock; a schedule S is *legal* under L if you can derive S by following L; finally, L *ensures* conflict serializability *iff* all legal schedules under L are conflict serializable

Two-Phase Locking

- *Two-phase locking* is a locking protocol that ensures serializability (as defined previously)
- The phases involved for a transaction *T* are:
  - *Growing phase* — *T* may obtain locks, but not release
  - *Shrinking phase* — *T* may release locks, but not obtain
- Following these two phases results in a *lock point* — the point in a transaction where it obtains its final lock
- Two-phase locking ensures serializability but does not completely avoid deadlock

Two-Phase Locking Variations

- *Strict two-phase locking:* exclusive locks must be held until *after* a transaction commits — this additional rule avoids cascading rollback
- *Rigorous two-phase locking:* instead of just exclusive locks, *all* locks must be held until after commit
- *Lock conversions:* without them, many transaction precedences result in serial schedules — we allow *upgrades* from shared to exclusive mode during the growing phase, then allow *downgrades* from exclusive to shared during the shrinking phase

Typical Locking Rules

- As you may have noticed, we haven't been using explicit lock requests in our SQL — the system determines this for us, based on the SQL involved
- Here's a typical algorithm:
  - A *read(Q)* becomes *lock-S(Q); read(Q);*
  - A *write(Q)* becomes *if (hasShared(Q)) then upgrade(Q); write(Q) else lock-X(Q); write(Q);*
  - Unlock everything only after the transaction either commits or aborts

Locking Odds & Ends

- The text includes a blurb on implementing a lock manager — useful to know, but only absolutely necessary if you're implementing a DBMS
- Two-phase locking is necessary and sufficient for ensuring serializability, and it doesn't require any additional information about a transaction
- However, if additional information is available, other protocols are possible — the text talks about *graph-based locking protocols* if you know the order in which transactions access data items

Timestamp-Based Protocols

- Another approach to concurrency control — execute transactions according to when they arrive at the server, vs. when conflicting locks are first acquired
- Assign a timestamp to each incoming transaction T_i (written as $TS(T_i)$), either via the system clock or a logical counter
- Assign two timestamps to each data item Q in the database: *W-timestamp(Q)* is the most recent successful *write(Q)*, and *R-timestamp(Q)* is the most recent successful *read(Q)*

- Use the following protocol:

```
if Ti requests read(Q) {
    if TS(Ti) < W-timestamp(Q) {
        rollback;
    } else {
        read(Q);
        R-timestamp(Q) = max(R-timestamp(Q), TS(Ti));
    }
}

if Ti requests write(Q) {
    if (TS(Ti) < R-timestamp(Q)) || (TS(Ti) < W-timestamp(Q)) {
        rollback;
    } else {
        write(Q);
        W-timestamp(Q) = TS(Ti);
    }
}
```

- Restarted transactions get a new timestamp
- Ensures conflict serialziability, avoids deadlock because waiting is not involved, does not avoid starvation if a long transaction keeps getting rolled back, requires some tweaks for recoverability and cascadelessness

Recoverable and Cascadeless Timestamp Protocol Tweaks

- Perform all writes at the end of a transaction, and do not allow transactions to read the data while they are being written
- Integrate limited locking: block transactions with incompatible or conflicting reads until the writing transaction commits successfully
- Track uncommitted writes: if a transaction read a data item that is written by other transactions before it, do not commit that transaction until the writing transactions have committed (recoverability only)

Thomas' Write Rule

- Observation: if $TS(T_i) < TS(T_j)$, and T_i writes Q before $TS(T_j)$, then we can just ignore that write, since it will be overwritten by T_j anyway
- *Thomas' write rule* is a protocol tweak on the *write(Q)* case based on that observation:

```
...
    if TS(Tᵢ) < W-timestamp(Q) {
       ignore write(Q);
    }
...
```

- Note how this is based on view serializability instead of conflict serializability

Validation-Based Protocols

- Split transactions into three phases: read, *validation*, and write — where validation is a check to see if a transaction's writes won't trample over others
- Each phase gets a timestamp, and $TS(T)$ is now the *validation timestamp* instead of its "arrival" timestamp; apply timestamp protocol with validation timestamp
- Considered to be *optimistic* instead of *pessimistic* — other protocols force a wait or rollback in *any* conflict, but this assumes that everything will be OK until validation time

Multiple-Granularity Locking

- Implementation/performance issue: allow multiple layers or scopes for different types of transactions
- Locks of different sizes: rows, tables, entire database
- A *granularity hierarchy* represents the different lock levels; a direct lock is *explicit*, while nodes underneath the explicit lock are *implicitly* locked
- Add an *intention lock mode* that locks nodes above an item that is explicitly locked
- The *multiple-granularity locking protocol* create a new compatibility function that inclodes intention locks

Multiversion Concurrency Control (MVCC)

- Maintain multiple *versions* of a database — each new *write(Q)* creates a new version of the data without immediately discarding the prior version
- Associate timestamps with versions: for version k of data item Q, *W-timestamp*(Q_k) corresponds to when version k was created, and *R-timestamp*(Q_k) is the timestamp of the last successful transaction to read Q_k
- Associate each transaction with the most recent Q_k that precedes it; all reads come from Q_k, and a *write(Q)* will only rollback if $TS(T) < R\text{-}timestamp(Q_k)$

- All versions older than the oldest transaction in the system except the most recent one can be discarded
- Reads will never fail in MVCC — matches well with the tendency of most databases to perform reads *way* more often than performing writes
- As is, does not ensure recoverability and cascadelessness; tweaks similar to those for timestamp protocols are required
- *Multiversion two-phase locking* tries to combine MVCC with two-phase locking: first, separate transactions to *read-only* and *update* categories; then, perform standard MVCC for read-only transactions and rigorous two-phase locking for update transactions, with update transactions updating a version counter

Deadlock Handling

- Through all this, we retained the open issue of deadlocks — note how we have said that deadlocks cannot be absolutely avoided in locking protocols
- Two primary approaches:
  - *Deadlock prevention* — try to avoid deadlock state in the first place
  - *Deadlock detection and recovery* — allow deadlocks, but catch and recover from them
- Issues include performance, overhead, rollback effects

Deadlock Prevention

- Different approaches, without a clear winner due to assorted tradeoffs:
- Lock all data items at one time — suboptimal if actual use of locked data items is sparse
- Lock items according to a specified data item order — prevents the locking "loops" that result in deadlock, but requires a sequencing scheme for data items
- Preemption and rollback: rollback transactions holding locks according to certain rules (*wait-die*, *wound-wait*)

Deadlock Detection

- Monitor transactions and their dependencies on each other through a *wait-for graph*
- Deadlock exists *iff* the wait-for graph has a cycle — all transactions in the cycle are considered deadlocked
- Detection procedure consists of:
  - Building and maintaining the current wait-for graph as transactions come in (i.e., create edges as locks are requested on already-locked items)
  - Periodically run the cycle detection algorithm

Deadlock Recovery

When a deadlock is discovered, recovery must take place — generally a rollback of one of the transactions:

- Choose the "victim" — involves a *cost function* for all transactions in the deadlock; kill the lowest cost T_i
- Perform the rollback — choose between *total rollback* (kill the entire transaction) or *partial rollback* (rollback only to the point of deadlock; requires additional system information)
- Watch for starvation — if a transaction repeats as the victim, make sure it doesn't repeat indefinitely

Concurrency Control Odds & Ends

Other topics tackled in the text:

- Special handling for *insert(Q)* and *delete(Q)* operations (special kind of *write(Q)* because they add or remove data instead of change something that exists)
- *Phantom phenomenon* — transactions that conflict not on a specific tuple, but over the tuples returned by a query (e.g., a sum() and an insert that affects the sum)
- *Weaker consistency* — allowing less consistency in the interest of better concurrency

PostgreSQL Specifics

- So now that you know all of these terms, how do they apply to a real-world system like PostgreSQL?
- As of PostgreSQL 8.1, concurrency control settings, behaviors, and features are:
  - MVCC by default
  - Almost-SQL-standard weakened consistency: *serializable* and *read committed* are supported
  - Explicit shared and exclusive locks, at multiple granularity levels