Introduction to Transactions

4010-441

Principles of Concurrent System Design

Transaction Background

- Originated in the database world.
- Very recently transferred to the internal memory world.
- Definition:

  A unit of work within a system to be treated in a coherent, correct and reliable way independent of other transactions.
- Basic purposes:
  - Provide reliability
    - Correct recovery from failures
    - Ensure system consistency
  - Provide isolation
    - No two threads / programs accessing the database should see "in-work" state of another thread / program.
    - Composite actions on the system's data are performed atomically.

Two Approaches

- Pessimistic
  - Lock all data items involved in the transaction:
    - May use read and write locks to distinguish role in the transaction.
    - Care on lock acquisition: Possibility of deadlock.
  - Perform the transformation.
  - Save the results.
  - Release the locks.
- Optimistic
  - Assume collisions are rare.
  - Duplicate items altered in local store (somehow).
  - Track items read but not written (somehow).
  - At end of transaction, perform a **commit** operation.
    - Success if no items used have changed since transaction start.
    - Abort if at least one item used has changed since transaction start.

ACID Properties of Transactions

Atomic

Either all of the operations in the transaction succeed or none of the operations persist.

Consistent

If the data are consistent before the transaction begins, then they will be consistent after the transaction finishes.

Isolated

The effects of a transaction that is in progress are hidden from all other transactions.

Durable

When a transaction finishes, its results are persistent and will survive a system crash.

Source: MSDN (http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx)

Atomicity

- AKA: Indivisibility or irreducibility.
- Prevents partial database / data structure update.
- Example #1: Booking seats on a set of connecting flights.
  - Successfully getting all but one flight is unacceptable.
  - Thus, the actions involved in getting the seats must be done as a unit.
- Example #2: Funds transfer
  - Withdrawal and deposit are a unit.
  - One but not the other is a windfall for one of the parties.
- Implementation:
  - Transaction code runs w/o locking.
  - Brief lock at the end to commit the result.
  - Or, if commit fails, rollback (undo) any temporary changes.

Consistency

- Predictable changes to system state.
- Usually with respect to distributed, replicated data.
- Contract between developer and system:
  - **IF** developer follows the rules.
  - **THEN** memory will be consistent and memory operations predictable.
- Example:
  - A database row is replicated on two nodes.
  - A client writes a new version of the row on node #1.
  - After time period ***t***, client B reads the row from node #2.
- Consistency model determines which row version B sees (preferably the updated one) and why.
- Example: the *happens after* relation in the JVM defines the consistency guarantees in the Java memory model.

Isolation

- Isolation levels: When do changes from a transaction become visible.
- Serializable (strongest guarantees)
  - W/ locks, any read or write locks are not released until the data are updated. Strongest guarantee, problematic performance.
  - W/O locks, locking done only at the end for a short burst. Write collisions are possible, leading to commit failures.
- Repeatable reads
  - Hold read & write locks throughout transaction.
- Read committed
  - Hold read locks until data initially accessed (may lead to different data if re-read).
- Read uncommitted
  - May see changes from uncommitted concurrent transaction.

Durability

- Permanent survival of committed transactions.
- If seat is booked, it remains booked even in face of failure.
- One approach:
  - When decide commit shall proceed, write the transaction log to non-volatile storage.
  - Then do the commit.
  - On failure, just play back the log.

ACID Failure Examples

Assume two integers, **X** and **Y** that must always sum to 100.

Atomicity failure

Subtract 10 from **X** but unable to access **Y** - if allowed through this violates atomicity (and the constraint).

Consistency failure

Assume transaction tries to add 1.5 to **X**, or to only change **Y**. Atomicity is not violated, but the constraint defining validity is.

Isolation failure

T_1 transfers 10 from **X** to **Y** and T_2 transfers 10 from **Y** to **X**, but the transactions are not isolated.

The possible race condition between T_1 and T_2 is a potential cause of an isolation failure.

Durability failure

Transaction tries to move 10 from **X** to **Y**. Subtract 10 from X, add 10 to Y, and report success. However, if the changes are in a memory buffer and power fails, the transaction is never completed.

Multi-version Concurrency Control

- Updates not performed by overwriting existing data.
- Instead, old data marked obsolete, and a new version is added.
- The data values themselves never change - only the identity of the "current" value.
- Provides "point in time" consistency:
  - Versions identified by sequence number or time stamp.
  - Fewer, shorter locks as only identity pointers (refs) must be changed.
- For a transaction:
  - Record the version of all objects it writes (possibly reads).
  - Changes are made to a "secret" copy of these objects.
  - At end of transaction, abort if versions of any referenced objects have changed.
  - Otherwise, atomically install the altered objects to define the next version.
- Note: This is the basic mechanism underlying Subversion.

Software Transactional Memory

- STM - An approach to simplifying concurrency.
- Early definition in Clojure (which actually emphasizes functional behavior and immutability).
- Clojure mutation philosophy:
  - *Identities*: Stable logical entity associated with different values over time.
  - Identities have a *state* at any point in time.
  - The state is an immutable *value*.
  - Even aggregates: I have a set of foods I like; if I change my preferences, this is a different set (not a change to the existing set).
- Clojure identities:
  - Refs to values (ref = fixed identity, value = current state associated with the identity).
  - Refs can change - but only under atomic guarantees.

Clojure Mutation

- Must be performed in a dosync
- (def balance (ref 0))
  - Identity is balance.
  - @balance is the value currently associated with identity balance.
- (ref-set balance 100)
  - Attempt to set balance to 100.
  - Fails - must ensure atomicity.
- (dosync

  ```
  (dosync
    (ref-set balance 100))
  ```
- (dosync

  ```
  (dosync
    (alter balance + 100)
    (alter balance - 50))
  ```

dosync vs synchronized

- Similar at first glance, but quite different
- STM encourages concurrency by letting threads compete fairly with other threads via wrapping in transactions.
- No explicit locking is done, no lock ordering, means deadlock free concurrency.
- Developer not responsible for designating blocks of code to be locked (synchronized).
- Concurrency driven by application behavior and data access.