Transactions

Transaction: Informal Definition

A transaction is a piece of code that accesses a shared database such that

- each transaction accesses shared data without interference from other transactions
- the code executes atomically, that is, if the transaction succeeds then all its updates are made permanent, but if the transaction fails for any reason, then none of its effects is left permanent

Example:

Consider a a database in a bank containing all customers' accounts

- each deposit and withdrawal executes as a transaction
- also, it's possible for a transaction to only read data (read-only transactions), e.g. checking the balance in some account

Example

Consider the act of withdrawing $100.00 from a particular account:

- the banking system would check if there is sufficient balance
- if not, then the transaction cannot continue (abort)
- else, subtract $100.00 from the current balance and store the value back in the database

From this example we can see:

- a need for executing the withdrawal without interference from operations that update this account (deposits or withdrawals) [same problems as in threads]
- a need for ensuring that, once the $100.00 is subtracted, that the database will reflect the results of the transaction permanently
- if a failure occurs in the middle, we would like to know that the account will be left in a consistent state (either the $100 is withdrawn or not, no in betweens)

Reality

Database operations are usually more complex than "subtract *x* from *y*":

- airline reservation systems (check availability, pricing, seating, etc. in one operation)
- inventory control (check out a set of items only if all of them are available, update inventor, coordinate with different sites, etc.)

Bottom line:

- the level of complexity increases very quickly
- the potential for failure increases
- transaction-based systems are not *allowed* to fail!

Why Transactions?

There are several alternatives:

- the banking database can use threads and conventional concurrency control to implement withdrawals and deposits, and perhaps the file system to implement durable storage
- the banking database can use processes and signaling to implement concurrency control and the file system for stable storage

But complexity:

- database can be spread across multiple files, or indeed, file systems
- operations to update the database may be written as separate programs from separate vendors (almost impossible to coordinate adhoc efforts)
- a myriad of failure scenarios require complex handling and recovery code that must be repeated for every operation

Transaction Syntax

A transaction is a syntactic structure:

begin-transaction

  ordinary program execution including

    reads from a database, writes to a database, and computations

commit or abort

Commit:

  An operation through which the transaction tells the system that it has succeeded and its effects should be made permanent

Abort:

  An operation through which the transaction tells the system that it has failed and its partial effects should be discarded (e.g. bank account does not contain sufficient funds for a withdrawal)

Failures and Recovery

All sorts of failures can occur

- hardware may fail
- O.S. may crash
- application may crash

Handling failures is through “discard & retry”

- such failures are modeled by “implicit” aborts for all active transactions at the time of failure
- it is up to the transaction processing system (TPS) to retry or rely on the user to retry (e.g. an ATM operation may fail with a message like “account is not available at this time, try later”, and the user will be expected to retry if she so chooses)

Transaction Discipline

A transaction may fail or abort at any point during execution:

- a transaction therefore cannot issue any output statement that cannot be rolled back (e.g. dispensing cache) until it actually commits
- transactions cannot communicate through input and output statements (i.e. all communication is implicit through the shared database)
- mathematically, it is impossible to atomically perform an output action with the commit operation (e.g. dispensing cache and the commit statement cannot be done atomically)
  - why? consider how to implement it. Are you going to release the output right after or right before the commit? Either way it doesn’t matter, because the mechanical device that performs the output statement can fail before or after the execution of the commit operation
- practically, it is done with various safeguards (e.g. limit amount of money you can withdraw in a single transaction, audit logs, etc.)

Transaction Properties

Transactions are often said to satisfy the ACID properties:

- A- atomicity: it's all or nothing. If the transaction commits, its effects are made visible in the database. If it aborts, its effects are discarded.
- C- consistency: the transaction reads a consistent view of the database and leaves it in a consistent state
- I- isolation: the transaction executes as if it is the only one in the system (therefore, there is no need for explicit concurrency control statements)
- D- durability: if a transaction commits, its effects are going to be in the database regardless of any present or future failures

Performance Considerations

Transaction systems must strive for best performance:

- throughput: number of transactions executed per second
- latency: average amount of time a transaction needs to finish

Satisfying the ACID properties with the above constraints excludes:

- the isolation property can be trivially achieved by executing one transaction at a time (very poor performance)
- the throughput can be improved by throwing more processors to the transaction processing system (cost factors prohibit this)

Often, a transaction is blocking to read data from disk, and the above two solutions will lead to obvious inefficiencies

Dilemma

An implementation of a transaction processing system must:

- provide concurrency control without explicit statements from the program
- provide high throughput and low latency despite disk I/O
- provide a way for ensuring atomicity without affecting the database due to uncommitted updates

None too easy, but:

- Serializability theory comes to rescue for concurrency control
- Logging comes to rescue for ensuring atomicity and durability of committed updates

Serializability

Given all the reads and writes from all active transactions, a scheduling of these operations is serializable if the schedule produces the same effect on the database as some serial execution of the same transactions.

Why does it help?

By definition, a serial execution of transactions does not have any concurrency control problem, since each transaction executes to completion before the next one is allowed to start

If we can find a serializable schedule, then the isolation property is satisfied

Serializability Challenges

Given a set of transactions that execute *concurrently*, the goal is:

- to find a schedule equivalent to some serial order
- maximize throughput and concurrency

Thus, exclude the trivial solution of running transactions one at a time.

But, theory is not perfect:

- given an arbitrary mix of reads and writes from different transactions, finding out all possible serializable orders is NP-complete

Example

Let the notation $r_i(x)$ denotes a read of item x by transaction i.

Let the notation $w_i(x)$ denotes a write of item x by transaction i.

Example: The following schedule is not serializable

$r_1(x)$, $r_2(x)$, $w_1(x)$, $w_2(x)$

Rules of thumb: study the relationship between transactions

- if a transaction T1 reads an item that is later written by T2, then T1 must precede T2 in any serializable order
- if a transaction T1 writes an item that is later read by T2, then T1 must precede T2 in any serializable order
- if a transaction T1 writes an item that is later written by T2, then T1 must precede T2 in any serializable order
- if a transaction T1 reads an item that is later read by T2 with no intervening writes, then there are no restrictions on ordering T1 & T2

Another Example

$r_1(x)$, $r_2(x)$, $w_1(x)$, $w_2(y)$, $r_1(y)$, $w_3(y)$
is equivalent to:
T2, T1, T3

$r_1(x)$, $r_2(x)$, $w_1(x)$, $r_2(y)$, $r_1(y)$, $r_3(y)$
is equivalent to:
T2, T1, T3, or
T3, T2, T1, or
T2, T3, T1

Practical Considerations

In reality:
- transactions are created dynamically all the time
- transactions do not often know which data items they will read or write
- a scheduler must order the reads and writes on the fly
  - acquisition phase

Need to produce serializable executions without scheduling overhead
Two-phase Locking
- a transaction scheduler acquires implicit locks for all data items read or written
- operation executes in two phases:
  - release phase

- Once a lock is released, no further locks are acquired

Practical Considerations (cont'd)

Typically locks are of two types:

- shared locks (reads)
- exclusive locks (writes)

Also:

Locks are usually held until the transaction commits or aborts:

- called strict two-phase locking
- prevents a situation called cascaded aborts, when a transaction reads the value written by a transaction that will abort in the future

Example:

$r_1(x)$, $r_2(x)$, $w_1(x)$, $w_2(y)$, $r_1(y)$, $w_3(y)$

This is equivalent to T2, T1, T3. But if T2 aborts, T1 will abort as well

Cascaded Aborts

An abort in one transaction may trigger the aborts of many:

How:

If a transaction T1 writes an item x, releases the lock, then
transaction T2 acquires a lock on x, reads it, releases the lock, then
transaction T3 acquires a lock on x, reads it, releases the lock, then
T1 aborts!

T2 should abort since it read a temporary value of x (atomicity violation)

Same for T3

Cascaded aborts are bad. Fix: Strict 2-phase locking

Recovery

Durability and atomicity are achieved through logging:

- No partial update is written to the database
- A transaction log is prepared when transaction starts
- All updates are reflected in the log
- Two types of logging:
  - redo logs -- save updates so that they can be applied to the home location of each item if a failure occurs after the transaction commits
  - undo logs -- save the pre-update values so that they can be restored if a failure occurs

in practice, use redo logs because of better performance & less complexity

Transaction Model