Distributed Systems (ICE 601)

Transactions & Concurrency Control - Part1

Dongman Lee
ICU

Transactions

- Definition
  - a sequence of one or more operations on one or more resources that is
    - **A**tomic: all or nothing
    - **C**onsistent: takes system from one consistent state to another
    - **I**solated: intermediate states invisible to others (serializable)
    - **D**urable: once completed (committed), changes are permanent
- Primitives
  - BeginTransaction
    - start transaction and get an ID
  - EndTransaction
    - commit (make all writes durable) or abort (discard all changes made by writes) transaction
  - AbortTransaction
  - Read, Write, ...

Class Overview

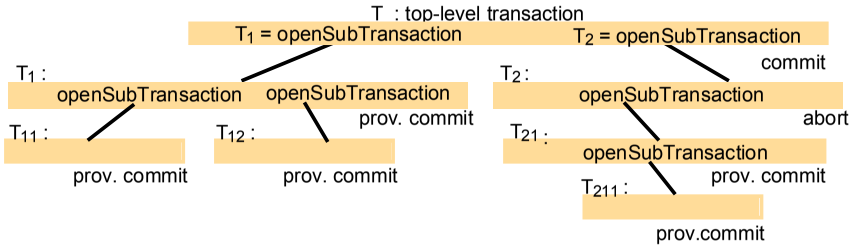
- Transactions
- Why Concurrency Control
- Concurrency Control Protocols
  - pessimistic
  - optimistic
  - time-based

Transactions (cont.)

- Issues with aborts
  - dirty reads
    - a transaction commits read operations on a value that another transaction wrote but aborted later
  - cascading aborts
    - all the related transactions abort together in a cascading fashion
  - premature writes
    - a value written by one transaction becomes nullified by the restored value that is restored by a recovery of other transaction after its abort
- Recoverability of transactions
  - a transaction that has a possibility of “dirty reads” should delay its commit until the affecting transaction commits
  - any read operation must be delayed until other transactions that applied a write operation to the same object have committed or aborted (stronger than recoverability)
  - write operations must be delayed until earlier transactions that

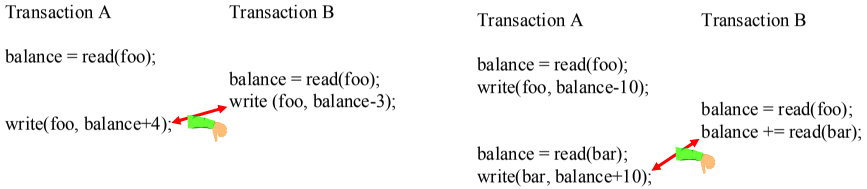
Nested Transactions

- Rules for commitment of nested transactions
  - a transaction may commit or abort only after its child transactions have completed
  - when a sub-transaction completes, it makes an independent decision either to commit provisionally or to abort. Its decision to abort is final
  - when a parent aborts, all of its sub-transactions are aborted
  - when a sub-transaction aborts, the parent can decide whether to abort or not



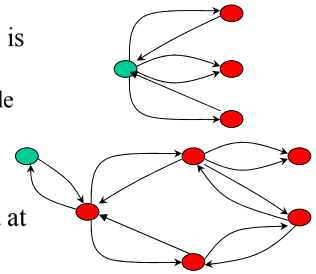
Why Concurrency Control?

- Concurrent access to a shared resource may cause inconsistency of the resource
  - inconsistency examples
    - lost updates
      - two transactions concurrently perform update operation
    - inconsistent retrievals
      - performing retrieval operation before or during update operation



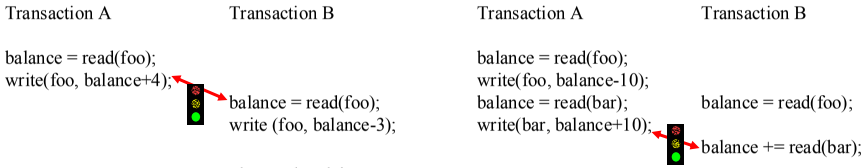
Distributed Transactions

- Definition
  - a transaction in which more than one server is involved
    - multiple servers are called by a client (simple distributed transaction)
    - a server calls another servers (nested transaction)
  - execution of program accessing shared data at multiple sites [Lamport]
- Requirement
  - a client requires to get congruent commitment from involved servers due to atomic property of a transaction
- Resolution
  - coordination



Basic Principle of Concurrency Control

- To avoid possible problems due to concurrent access, operations of related transactions must be *serialized (one-at-a-time)*



  - *strict* two-phase locking
    - lock is obtained (phase 1) before operations and released (phase 2) after the transaction commits or aborts
    - granularity is too big!

➲ concurrency control protocols

Concurrency Control Protocols

- Read and Write operation conflict rules

Transaction A	Transaction B	Conflict	Reason
Read	Read	No	No dependency
Read	Write	Yes	Depends on execution order
Write	Write	Yes	Same as above

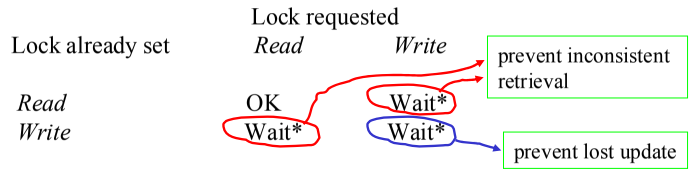
- Three approaches
  - Locking
  - Optimistic method
  - Timestamp ordering

Locking (cont.)

- Lock promotion
  - to escalate the level of exclusiveness
  - rules
    - promote a read lock to a write lock when the transaction attempts to update the data that it has retrieved
    - if a read lock is shared, it can't be promoted; instead, request a write lock
- Lock manager
  - responsible for managing a table of locks each entry of which includes
    - transaction id
    - data id
    - lock type
    - condition variable

Locking

- Two types of locks
  - read locks: shared locks
    - more than one transaction can share it
  - write locks: exclusive locks
    - one at a time
    - wait until the lock is released
- Operation conflict rules



* wait until the transaction commits or aborts

Locking (cont.)

- Locking rules for nested transactions
  - Locks set by children are inherited by their parents
  - Parents are not allowed to run concurrently with their children
  - Sub-transactions at the same level are allowed to run concurrently
  - When a subtransaction acquires a read lock on an object, no other transaction except only its parent can get a write lock on the same object
  - When a subtransaction acquires a write lock on an object, no other transaction except only its parent can get a read or write lock on the same object
  - When a subtransaction commits, its locks are inherited by its parent
  - When a subtransaction aborts, its locks are discarded but its parent continue to retain the locks if the parent already has them

Locking (cont.)

- Two-version locking [Gifford]
  - allows more concurrency by *deferring write locks till commit time*
    - read operations are allowed while write operation is being performed
      - write operation is done on a tentative version of data items
      - read operation is done on committed version
  - three types of locks: read, write, & commit locks

	Lock to be set		
Lock already set	*Read*	*Write*	*Commit*
Read	OK	OK	Wait
Write	OK	Wait	-
Commit	Wait	Wait	-

  - vs. ordinary read-write locking
    - pro: read operation is only delayed during commit phase instead of entire phases
    - con: read operation can cause delay in committing other transactions

Locking (cont.)

- Problems in locking-based concurrency control
  - extra overhead to manage locking which may not be required
  - use of lock can give a rise to *deadlock*
  - locks cannot be leased until the end of the transaction to avoid cascading aborts

Locking (cont.)

- Hierarchic locks[Gray]
  - allows mixed granularity locks, building a hierarchy of locks
    - giving owner of lock explicit access to node in hierarchy and implicit access to its children
  - introduces an additional type of lock: intention-Read/Write
    - before a child node is granted a read/write lock, an intention to read/write lock is set on the parent node

	Lock to be set			
Lock already set	*Read*	*Write*	*I-Read*	*I-Write*
Read	OK	Wait	OK	Wait
Write	Wait	Wait	Wait	Wait
I-Read	OK	Wait	OK	OK
I-Write	Wait	Wait	OK	OK

  - vs. ordinary read-write locking
    - pro: reduce # of locks when mixed granularity locking is required

Deadlocks

- Definition
  - a state in which each member of a group of transactions awaits some other member to release a lock
    - examples
      - transactions T and U read the data and both try to promote their read lock to write lock
      - transaction T waits for transaction U to release a lock on a data item A while transaction U waits for transaction V to release a lock on a data item bar and transaction V waits for transaction T to release a lock on a data item C
- Wait-for-graphs
  - graphical notation to represent wait-for relations among transactions

Deadlocks (cont.)

- Deadlock prevention
  - locks all of the data items at the beginning
    - hard to predict all the required data items at the beginning
  - requests locks on data items in a predefined order
    - may result in premature locking and reduction in concurrency
- Deadlock detection
  - lock manager checks deadlocks
    - whenever a lock request from a transaction is given to a data item currently locked by another transaction, or
    - less frequently to avoid server overhead
  - lock manager does the following operations to detect a deadlock
    - finds a cycle in the wait-for-graph, and
    - break the cycle
  - once detected, one of transactions in a cycle is selected and aborted based on age and # of cycles it gets involved
- Deadlock resolution
  - once detected, one of transactions in a cycle is selected and aborted
  - timeouts