## ACID Properties of a Transaction (Review)

- **A**tomicity — a transaction is either performed in its entirety or not at all; it appears to an outside observer as a single, instantaneous, indivisible action
- **C**onsistency — a transaction must take the database from one consistent state to another; invariants that should always hold will hold after the transaction
- **I**solated (Serializable) — if two transactions run at the same time, the result must look as if they ran sequentially in some arbitrary order; a transaction's updates must not be visible to other transactions until it commits
- **D**urable — once a transaction commits, its result is permanent (must never be lost)



## Need for Concurrency Control (Review)

- Lost update problem:

| Transaction T | | Transaction U | |
|---|---|---|---|
| bal=read(A) | \$100 | | |
| write(A,bal–4) | \$96 | | |
| | | bal=read(C) | \$300 |
| | | write(C,bal–3) | \$297 |
| bal=read(B) | \$200 | | |
| | | bal=read(B) | \$200 |
| | | write(B,bal+3) | \$203 |
| write(B,bal+4) | \$204 | | |

- Inconsistent retrievals problem:

| Transaction T | | Transaction U (part) | |
|---|---|---|---|
| bal=read(A) | \$200 | | |
| write(A,bal–100) | \$100 | | |
| | | bal=read(A) | \$100 |
| | | bal+=read(B) | \$300 |
| bal=read(B) | \$200 | | |
| write(B,bal+100) | \$300 | | |



## Why do These Problems Occur?

- *Conflicts* between transactions cause this inconsistency due to the order in which the operations are executed
  - If one transaction reads a data object, and another reads that same data object, there is not a conflict
  - If one transaction reads a data object, and another writes that same data object, there is a conflict
  - If one transaction writes a data object, and another writes that same data object, there is a conflict
- It's up to some *concurrency control* mechanism to allow interleaving, but keep the database / file consistent
  - Should allow high degree of concurrency
  - Should prevent intermediate values from being visible to other transactions



## Issues in Transactions and Concurrency Control

- Centralized transactions
  - Concurrency control
    - Locking algorithms
      - Static locking
      - Two-phase locking (2PL)
      - Strict two-phase locking (strict 2PL)
    - Optimistic concurrency control
    - Timestamp ordering
  - Handling deadlock for locking algorithms
    - Deadlock detection
    - Deadlock prevention
      - Lock timeouts
      - Transaction timestamps
- Distributed transactions
  - Simple distributed vs. nested
  - Atomic commit protocols
    - One-phase
    - Two-phase

## Concurrency Control Using Locks (Eswaran, Gray, Lorie, and Traiger, 1976)

- A *well-formed* transaction must:
  - Lock a data object before accessing it
  - Unlocks the data object before it completes (commit / abort)
  - Example:
    lock B; read B; update B; unlock B
- Note that being well-formed is ***not*** sufficient to guarantee serializabilty
  - Well-formed doesn't say anything about ***when*** a transaction should lock / unlock
    - Lock sometime after transaction begins, but before object is accessed
    - Unlock after finished with object, but before transaction completes
  - Additional constraints are needed to specify when a lock can be acquired, and when it can be released
    - These constraints are expressed as *locking algorithms*



## Static Locking

- A transaction acquires locks on ***all*** the data objects it needs (at a single point in time) before executing ***any*** action on the data objects
  - Usually when transaction begins
- After using the data objects, it releases ***all*** of its locks at once
  - Usually when transactions completes, else intermediate values will be visible
- Evaluation:
  - ✔ Simple, yet preserves consistency (intermediate values are not visible to other transactions)
  - ✘ Requires *a priori* knowledge of all the data objects to be accessed
  - ✘ Wasteful of resources, severely limits the concurrency of the transactions



## Two-Phase Locking (2PL)

- A transaction acquires a lock when it needs to access a data object. If it releases the lock after that access, but before the transactions ends, data could become visible to other transactions
  - ➥ (Consistency constraint) A transaction cannot request a lock on any data object after it has unlocked a data object
- The algorithm has two phases:
  - *Growing phase* — transaction requests locks, but doesn't release any locks
    - The stage of a transaction when it holds locks on all the needed data objects is called the *lock point*
  - *Shrinking phase* — transaction releases locks, but doesn't request any more locks
- Increases concurrency over static locking because locks are held for less time



## Two-Phase Locking (2PL) (cont.)

- Problems with two-phase locking (2PL):
  - Prone to *cascaded roll-back*
    - With 2PL, after the transaction has released some of its locks, yet before it has committed the transaction, those intermediate results become visible
    - When a transaction is rolled back, all modified data objects are restored
    - What if another transaction reads those intermediate results, and this transaction later aborts?
      - All transactions that have read these data objects must also be rolled back (even if they've already completed!) — this is called *cascaded roll-back*
  - Prone to deadlock
    - A transaction can request a lock on a data object while holding locks on other data object, so a circular wait can result
    - Resolved (after detecting deadlock) by:
      - Abort deadlocked transaction, restore all modified data objects, release all its locks, and withdraw all pending lock requests

## Improvements to Two-Phase Locking

- Strict two-phase locking (strict 2PL)
  - A transaction holds all its locks until it completes, when it commits and releases all of its locks in a single atomic action
    - Similar for an abort
  - ✘ Reduces concurrency (transactions hold locks longer than in 2PL) — almost as bad as strict locking!
  - ✘ Doesn't avoid deadlock
  - ✔ Avoids cascaded roll-backs
  - Most common locking algorithm
- Improvements to these algorithms
  - Two kinds of locks:
    - Read lock — other readers are permitted, writers are excluded
    - Write lock — exclusive access
  - Reduce granularity where possible (more concurrency, also more locks)



## Deadlock Detection / Prevention for Locking Algorithms

- Deadlock detection
  - Lock manager is responsible for detection
    - It looks for cycles in its WFG
    - If it finds a cycle, it must select and abort a transaction
- Deadlock prevention
  - Lock all items when transaction starts
    - Overly restrictive, reduces concurrency
    - May not be possible to predict accesses
  - Request locks in predefined order
    - May cause premature locking, which reduces concurrency
  - Lock timeouts (enables preemption)
    - Each lock is invulnerable for a limited period, and vulnerable afterwards
    - If a transaction wants to access a data object protected by a vulnerable lock, the lock is broken and the transaction holding it is aborted



## Deadlock Detection / Prevention for Locking Algorithms (cont.)

- Deadlock prevention (cont.)
  - Transaction timestamps
    - Each transaction is assigned a unique timestamp when it starts (logical clock, using Lamport's algorithm)
    - If a transaction needs to access a data object that is locked by another transaction, the timestamps of the two transactions are compared
      - Older transaction (smaller timestamp) generally have priority
      - Wait-for edges are only allowed from older to younger, which prevents cycles
    - Wait-die: (non-preemptive)
      - If older transaction wants something held by younger transaction, it <u>waits</u>
      - If younger transaction wants something held by older transaction, it must <u>die</u>
    - Wound-wait: (preempts resource)
      - If older transaction wants something held by younger transaction, it <u>preempts</u> it (wounds it)
      - If younger transaction wants something held by older transaction, it <u>waits</u>



## Optimistic Concurrency Control (Kung and Robinson, 1981)

- Disadvantages of locking:
  - High lock maintenance overhead
    - Even read-only queries must lock
  - Possible deadlock and cascading aborts
    - Deadlock prevention reduces concurrency
    - Holding locks until the end to prevent cascading aborts reduces concurrency
- Alternative — optimism
  - Likelihood of conflict is low, so just ignore the problem for the most part
    - Allow transactions to proceed as if there is no possibility of conflict
    - Use private workspaces
  - Validation before closing — if none of the data objects were modified by other transactions, then the transaction can commit, otherwise it aborts
  - No deadlock, no cascading aborts

## Timestamp Ordering

- Each operation is validated when it is carried out
  - If it can not be validated, then the entire transaction is aborted
- Basic timestamp ordering algorithm:
  - Each transaction is assigned a unique timestamp when it starts (logical clock, using Lamport's algorithm)
  - A transaction's request to <u>write</u> a data item is valid only if that data item was last <u>read and written</u> by earlier transactions
  - A transaction's request to <u>read</u> a data item is valid only if that data item was last <u>written</u> by earlier transactions
  - If a transaction is aborted and restarts, it gets a new timestamp
  - No deadlock, no cascading aborts



## Comments on the Various Concurrency Control Methods

- Pessimistic
  - Two-phase locking and timestamp ordering are both pessimistic — detect conflicts as each data item is accessed
  - Static vs. dynamic ordering
    - Timestamp ordering decides serialization order statically — when each transaction starts
    - Two-phase locking decides serialization order dynamically — according to the order in which the data items are accessed
- Effect of conflict:
  - Timestamp ordering aborts immediately
  - Two-phase locking makes transaction wait
  - Optimistic concurrency lets all transactions proceed, but later aborts some (possibly after long execution)



## Distributed Transactions

- A *distributed transaction* invokes operations in several different servers
  - Simple distributed transaction
    - Client makes requests to more than one server
    - Each server carries out the client's requests without involvement by others
  - Nested distributed transaction
    - Client makes requests to more than one server
    - Some of those servers make requests of yet other servers to carry out the client's request, and some of those servers may…
    - Example:
      - Client A tells server M to transfer $4 from account A to C, and $3 from B to D
      - A is at server X, B is at server Y, and C and D are at server Z
      - M tells server X to withdraw $4 from A
      - M tells server Y to withdraw $3 from B
      - M tells server Z to deposit $4 into C, and $3 into D



## Atomic Commit Protocols

- Distributed transactions are still required to be completed atomically
- First server involved in the distributed transaction becomes the coordinator
  - Coordinator is responsible for committing or aborting the transaction
  - All transactions involved know the identity of the coordinator
- One-phase atomic commit protocol
  - Transaction ends when coordinator requests that it be committed or aborted
  - Coordinator tells all the servers in the transaction to commit / abort, and keeps repeating that request until all of them acknowledge that they have carried it out
  - Coordinator can commit / abort, but individual servers can not

## Atomic Commit Protocols (cont.)

- Two-phase atomic commit protocol
  - Allows any server to abort its part of the transaction; atomicity then requires the entire transaction to be aborted
  - Phase 1: (voting phase)
    - Coordinator asks each worker if it can commit its transaction
    - Worker replies to coordinator; if its answer is *no*, the worker immediately aborts
  - Phase 2: (completion phase)
    - Coordinator collects the votes (including its own)
      - If there are no failures, and all votes are *yes*, the coordinator sends a *commit* request to each worker
      - Otherwise, the coordinator sends an *abort* request to all workers that voted *yes*
    - Workers that voted *yes* wait for a *commit* or *abort* message, act accordingly, and in the case of *commit* send a have_*committed* message afterwards