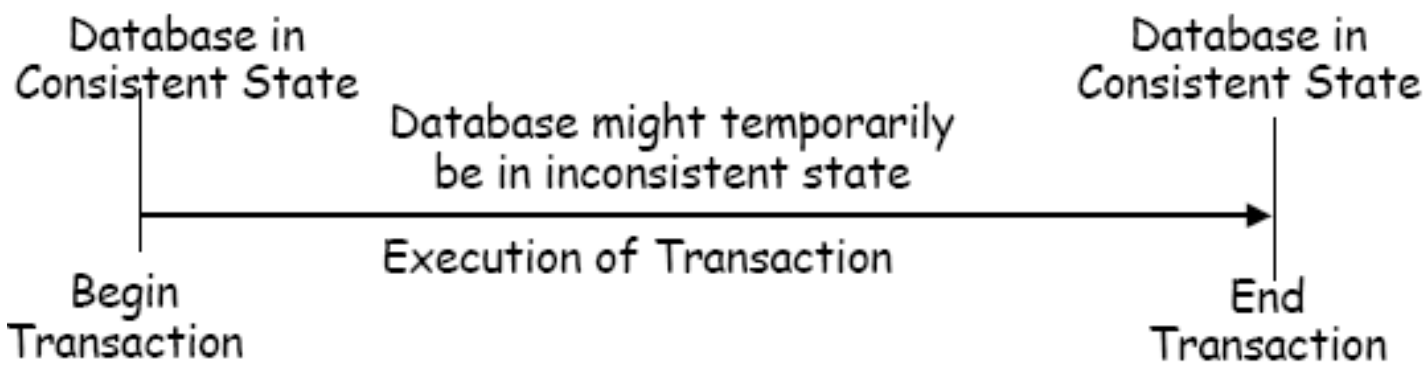
Transactions

CMPT 431

Transactions

- A transaction is a collection of actions logically belonging together
- To the outside world, a transaction must appear as a ***single indivisible operation***



Use of Transactions In Distributed Systems

- **Correct concurrent operations**
  - Example: updating the bank account
    - [1] Read the current balance in the account
    - [2] Compute the new balance
    - [3] Update the database to record the new balance
  - On concurrent access, operations done by multiple threads may interleave
  - This leads to incorrect operation
  - Transactions ensure proper operation
- **Masking failures**
  - In a replicated bank database – the account is updated at two sites
  - One site is updated, the other one crashes before the update is complete
  - The system's state is ***inconsistent***
  - Transactions ensure proper recovery

ACID Properties of Transactions

- ***Atomicity*** – transaction is indivisible – it completes entirely or not at all, despite failures
- ***Consistency*** –system rules (invariants) are maintained despite crashes or concurrent access
- ***Isolation*** – transactions appear indivisible to each other despite concurrent access
  - If multiple threads execute transactions the effect is the same as if transactions were executed sequentially in some order
- ***Durability*** – effects of committed transactions survive subsequent failures

Maintaining ACID Properties

- To maintain ACID properties, transaction processing systems must implement:
  - Concurrency control (Isolation, Consistency)
  - Failure Recovery (Atomicity, Durability)

Concurrency Control

- Implemented using locks (or other synchronization primitives)
- A naïve approach: one global lock – no transactions can proceed simultaneously. ***Bad performance***
- A better approach: associate a lock with each data item (or group of items)
- Acquire locks on items used in transactions
- Turns out ***how*** you acquire locks is very important

Concurrency Control

- **Transaction 1**

  lock(x)

  update (x)

  unlock (x)

  …

  abort

- **Transaction 2**

  …

  lock(x)

  read(x)

  unlock (x)

  commit

saw inconsistent data!

Strict Two-Phase Locking

- <u>Phase 1: </u>A transaction can acquire locks, but cannot release locks
- <u>Phase 2: </u>A transaction releases locks ***at the end*** – when it aborts or commits

Non-Strict Two-Phase Locking

- Allows releasing locks ***before*** the end of the transaction, but ***after the transaction acquired all the locks*** it needed
- Often impractical, because:
  - We do not know when the transaction has acquired all its locks – lock acquisition is data dependent
  - Cascading aborts: early lock release requires aborting all transactions that saw inconsistent data "If I tell you this, I'll have to kill you"

Deadlock

- When we acquire more than one lock at once we are prone to ***deadlocks***
- Techniques against deadlocks:
  - Prevention
  - Avoidance
  - Detection
- **Prevention:** lock ordering. <u>Downside</u>**:** may limit concurrency. Locks are held longer than necessary
- **Avoidance:** if a transaction has waited for a lock for too long, abort the transaction. <u>Downside:</u> transactions are aborted unnecessarily
- **Detection:** wait-for graph (WFG) – who waits for whom. If there is a cycle, abort a transaction in a cycle. <u>Downside</u>: constructing WFGs is expensive in distributed systems.

Maintaining ACID Properties

- To maintain ACID properties, transaction processing systems must implement:
  - Concurrency control (Isolation, Consistency)
  - **Failure Recovery (Atomicity, Durability)**

Types of Failures

- **Transaction abort** – to resolve deadlock or as requested by the client
- **Crash** – loss of system memory state. Disk (or other non-volatile storage) is kept intact
- **Disk failure**
- **Catastrophic failure** – memory, disk and backup copies all disappear

We will discuss these in detail

Abort Recovery

- Accomplished using ***transactional log***
- Log is used to “remember” the state of the system in case recovery is needed
- How log is used depends on update semantics:
  - <u>In-place updates </u>(right away)
  - <u>Deferred updates </u>(at the end of transaction)

Transactions With In-Place Updates

- Update: record an ***undo record*** (e.g., the old value of the item being updated) in an ***undo log***, and update the database
- Read: simply read the data from the database
- Commit: flush database changes to disk, ***then*** discard undo records
- Abort: Use the undo records in the log to back out the updates

Transactions with Deferred Updates

- Update: Record a ***redo record*** (e.g., the new value of the item being updated) in a redo log
- Read: combine the redo log and the database to determine the desired data
- Commit: Update the database by applying the redo log in order, flush the log to disk, ***then*** report successful commit

  Here commit needs not flush the database to disk, just the log
- Abort: do nothing

Crash Recovery

- A crash may leave the database inconsistent
  - The database may contain data from uncommitted or aborted transactions
  - The database may lack updates from committed transactions
- After the crash we would like
  - Remove data from uncommitted or aborted transactions
  - Re-apply updates from committed transactions

Recovery With Undo Logging

- All committed transactions would have been flushed to disk, so no need to redo them
- Use undo records to remove data from uncommitted or aborted transactions
- What if an update was written to database ***before*** the undo record was written to log?
- ***Write-ahead log rule:*** *A undo record must be flushed to disk before the corresponding update is reflected in the database*

Recovery with Redo Logging

- No uncommitted or aborted transactions would have been in the database, so no need to undo them.
- Redo all updates for committed transactions (use redo records).
- Redo records must be idempotent, in case we crash during recovery
- What if the client committed transaction, but the system crashed ***before*** "commit" log record made it to disk?
- ***Redo rule:*** *We must flush the commit record to disk before telling the client that transaction is committed*

Performance Considerations: Disk Access

- Each transaction necessarily involves disk access (expensive)
- To reduce performance costs, log is kept on a separate disk than database
- Log is written sequentially under normal operation
- Sequential writes are fast
- That is why redo logging is better for performance, since you don't have to flush the database to disk on commit
- Database is updated asynchronously, pages are *eventually* flushed to disk, so it's not a performance bottleneck

Performance Considerations: Log Size

- Log will keep on growing forever…
- To prevent this, we use checkpoints
- If the data has been flushed to the database disk, discard corresponding commit records
- For each transaction, keep a log sequence number (LSN)
- In the checkpoint record, record the smallest LSN of all active transactions
- Discard undo records with LSN below the current smallest LSN

Summary

- Transactions are used for concurrent operations and for masking failures
- ACID properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- To maintain these properties, databases implement:
  - Concurrency control (two-phase locking, deadlock resolution)
  - Failure recovery (logging, redo/undo)