Transactions and 2-phase-commit

Jinyang Li
What we’ve learnt so far...

• Consistency semantics for single-op, single-object access
  – linearizability, sequential consistency
  – eventual/causal consistency, per-object sequential consistency

• Providing failure tolerance for linearizable storage
  – Paxos, Raft, Viewstamp replication
Today’s topic: transactions

• Application perform multi-operation, multi-object data access
  – Transfer money from one account to another
  – Insert Alice to Bob’s friendlist and insert Bob to Alice’s friendlist.

• What if?
  – Failures occurs in the middle of writing objects
  – concurrent operations race with each other?
ACID transactions

• **A** (Atomicity)
  – All-or-nothing w.r.t. failures

• **C** (Consistency)
  – Transactions maintain any internal storage state invariants

• **I** (Isolation)
  – Concurrently executing transactions do not interfere

• **D** (Durability)
  – Effect of transactions survive failures
The Recovery Challenge

T1: Transfer $100 from A to B

- \( x := \text{Read}(A) \)
- \( y := \text{Read}(B) \)
- Write(A, x-100)
- Write(B, y+100)

- **A** T1 fully completes or leaves nothing
- **D** once T1 commits, T1’s writes are not lost
- **I** no races, as if T1 happens either before or after T2
- **C** preserves invariants, e.g. account balance > 0
Solution: WAL logging

- **Write-Ahead-Logging (WAL)**
  - All state modification must be written to log before they are applied

- **Simplest WAL: REDO logging**
  - Only stores REDO information in log entries
  - transactions buffer writes during execution
    - This requirement is easy to satisfy now, but not the case in 80s/90s when memory capacity is very low
Example using REDO-log

T1: transfer $100

x := Read(A)  // x=300
y := Read(B)  // y=200
Write(A, x-100)  // A←200
Write(B, y+100)  // B←300
Commit

T1 issues commit

system flushes log to durable storage

system appends T1’s entry to log

A←200 is written to global storage

B←300 is written to global storage

T1’s commit returns

.... T0: A←300  T1: A←200, B←300
Example using REDO-log

System state at recovery

Log contains F1, T0, T1
Latest checkpoint state F1:
A=0, B=200

REDO T0

Global state:
A=300, B=200

REDO T1

Global state:
A=200, B=300

T0: A<->300

T1 issues commit
system appends T1’s entry to log

A<->200 is written to global storage

B<->300 is written to global storage

T1’s commit returns

checkpoint F1
The Concurrency Control Challenge

T1: Transfer $100 from A to B

T2: Transfer $100 from A to C

- **A**  T1 completes or nothing (ditto for T2)
- **D**  once T1/T2 commits, stays done, no updates lost
- **I**  no races, as if T1 happens either before or after T2
- **C**  preserves invarants, e.g. account balance > 0
Concurrency control challenge: problematic interleaving

T1: Transfer $100 from A to B
x := Read(A)
y := Read(B)
if x > 100 {
    Write(A, x-100)
    Write(B, y+100)
    Commit
} else {
    Abort
}

T2: Transfer $50 from A to C
x := Read(A)
y := Read(C)
if x > 50 {
    Write(A, x-50)
    Write(C, y+50)
    Commit
} else {
    Abort
}
Ideal isolation semantics: serializability

• Definition: execution of a set of transactions is equivalent to some serial order
  – Two executions are equivalent if they have the same effect on database and produce same output.
Conflict serializability

- An execution schedule is the ordering of read/write/commit/abort operations

\[
\begin{align*}
x &= \text{Read}(A) \\
y &= \text{Read}(B) \\
\text{Write}(A, x+100) \\
\text{Write}(B, y-100) \\
\text{Commit}
\end{align*}
\]

\[
\begin{align*}
x &= \text{Read}(A) \\
y &= \text{Read}(B) \\
\text{Print}(x+y) \\
\text{Commit}
\end{align*}
\]

A (serial) schedule: \text{R(A),R(B),W(A),W(B),C,R(A),R(B),C}
Conflict serializability

• Two schedules are equivalent if they:
  – contain same operations
  – order conflicting operations the same way

• A schedule is serializable if it’s equivalent to some serial schedule

• Strict serializability / Order-preserving serializability
  – If T finishes before T’ starts, T must be ordered before T’ in equivalent serial schedule

Two ops conflict if they access the same data item and one is a write.
Serializability Example

 Serializable?

\[
R(A), R(B), \underline{R(A)}, \underline{R(B)}, C \underline{W(A)}, \underline{W(B)}, C
\]

Equivalent serial schedule:

\[
R(A), R(B), C \underline{R(A)}, \underline{R(B)}, W(A), \underline{W(B)}, C
\]
Examples

T1
x = Read(A)
y = Read(B)
Write(A, x-100)
Write(B, y+100)

T2
x = Read(A)
y = Read(B)
Print(x+y)

Serializable? R(A), R(B), W(A), R(A) R(B), C W(B) C
Examples

 Serializable?
R(A), R(B), W(A), R(A) R(C), C R(B) W(B), C W(C) C

T1
x = Read(A)
y = Read(B)
Write(A, x-100)
Write(B, y+100)

T2
x = Read(A)
y = Read(C)
Print(x+y)

T3
x = Read(B)
Write(C, x)

W(A) \rightarrow R(A)
R(B) \rightarrow W(B)
R(C) \rightarrow W(C)
Realize a serializable schedule

• Locking-based approach

• Strawman solution 1:
  – Grab global lock before transaction starts
  – Release global lock after transaction commits

• Strawman solution 2:
  – Grab short-term fine-grained locks on an item before access
  – Lock(A) Read(A), Unlock(A), Lock(B) Write(B), Unlock(B) ....
Strawman 2’s problem

Possible? (short-term, fine-grained locks on reads/writes)
R(A), R(B), W(A), R(A), R(B) C W(B) C

T1
x = Read(A)
y = Read(B)
Write(A, x-100)
Write(B, y+100)

T2
x = Read(A)
y = Read(B)
Print(x+y)

Locks on writes should be held till end of transaction

Read an uncommitted value
More Strawmans

• Strawman 3
  – fine-grained locks
  – long-term locks for writes
    • grab lock before write, release lock after tx commits/aborts
  – short-term locks for reads
Strawman 3’s problem

Possible? long-term locks for writes, short-term locks for reads
R(A), R(B), W(A), R(A), R(B), C W(B), C

Read locks must be held till commit time

Non-repeatable reads

R(A), R(A), R(B), W(A), W(B), C R(B)
Realize a serializable schedule

- 2 phase locking (2PL)
  - A growing phase in which the transaction is acquiring locks
  - A shrinking phase in which locks are released

- In practice,
  - The growing phase is the entire transaction
  - The shrinking phase is at the commit time

- Optimization:
  - Use read/write locks instead of exclusive locks
2PL in practice: an example

RLock(A)
x = Read(A)
RLock(B)
y = Read(B)
WLock(A)
buffer A ← x - 100
WLock(B)
buffer B ← y + 100
T1 issues commit:
log (A ← 0, B ← 200)
Write(A, 0)
Unlock(A)
Write(B, 200)
Unlock(B)
Commit returns

Print(x + y)
Unlock(A)
Unlock(B)

RLock(A)
x = Read(A)
RLock(B)
y = Read(B)
Print(x + y)
Unlock(A)
Unlock(B)
More on 2PL

• What if a lock is unavailable? wait
• Deadlocks possible?
• How to cope with deadlock? detect & abort
  – Grab locks in order? No always possible
  – Transaction manager detects deadlock cycles and aborts affected transactions
  – Alternative: timeout and abort yourself
How to support distributed transactions?

• Storage is sharded across multiple machines
  – Different machines store different subset of data

• Challenge: machine failures
• What can go wrong?
  – A does not have enough money
  – Node B has crashed
  – Coordinator crashes
  – Some other client is reading or writing to A or B

Client transaction

\[
\begin{align*}
A & := A-100 \\
B & := B+100
\end{align*}
\]
Reasoning about correctness

• TC, A, B each has a notion of committing
• Correctness:
  – If one commits, no one aborts
  – If one aborts, no one commits
• Performance:
  – If no failures, A and B can commit, then commit
  – If failures happen, find out outcome soon
Correctness first

If $r_A = \text{yes}$ && $r_B = \text{yes}$
    outcome = “commit”
else
    outcome = “abort”

Commits upon receiving “commit”, unlock A
Performance Issues

• What about timeouts?
  – TC times out waiting for A’s response
  – A times out waiting for TC’s outcome message

• What about reboots?
  – How does a participant clean up?
Handling timeout on A/B

- TC times out waiting for A (or B)’s “yes/no” response
- Can TC unilaterally decide to commit?
  - no
- Can TC unilaterally decide to abort?
  - depends. In traditional 2PC, yes.
Handling timeout on TC

• If A or B responded with “no” …
  – Can either unilaterally abort?

• If both A and B responded with “yes”
  – Can they unilaterally abort?
  – Can it unilaterally commit?
Traditional 2PC is not failure-tolerant

- If TC can unilaterally abort
  - System blocks if TC fails and both A/B voted “yes”.
- If TC cannot unilaterally abort
  - System blocks if either A or B fails
Recovery upon reboot

- TC logs “commit” on disk before replying to client
- A/B logs “yes” vote on disk before replying 2PC-prepare

Recovery:
  - If TC finds no “commit” on disk, abort
  - If TC finds “commit”, commit
  - If A/B finds no “yes” on disk, abort
  - If A/B finds “yes”, asks TC for transaction status
A Case study of 2P commit in real systems

Sinfonia (SOSP’ 07)
What problem is Sinfonia addressing?

• Targeted uses
  – systems or infrastructural apps within a data center

• Sinfonia: a shared data service
  – Span multiple nodes
  – Replicated with consistency guarantees

• Goal: reduce development efforts for system programmers
Sinfonia architecture
Sinfonia mini-transactions

• Provide a restricted form of ACID transactions
  – as well as before-after atomicity (using locks)

• Trade off expressiveness for efficiency
  – fewer network roundtrips to execute
  – Less flexible, general-purpose than traditional transactions
Mini-transaction details

- Mini-transaction
  - Check compare items
  - If match, retrieve data in read items, modify data in write items

- Example (atomic-swap):

```java
t = new Minitransaction()
t->cmp(A, 3000)
t->cmp(B, 2000)
t->write(A, 2000)
t->write(B, 3000)
Status = t->exec_and_commit()
```
Mini-transaction vs. Traditional Distributed Transaction

**Traditional transactions:**
- General but expensive
- BEGIN tx
  - x = Read(A)
  - y = Read(B)
  - Write(A, y)
  - Write(B, x)
  - END tx

**Mini-transaction:**
- Less general but efficient
- BEGIN tx
  - If (a == 3000 && b == 2000)
    - a = 2000
    - b = 3000
  - END tx
Sinfonia’s 2P protocol

• Transaction coordinator is at application client instead of memory node
  – Saves one RTT

• TC cannot unilaterally abort
  – Because application clients are less reliable and they do not keep logs
Sinfonia’s 2P protocol

• A transaction is committed iff all participants have “yes” in their logs
• Recovery coordinator cleans up
  – Ask all participants for existing vote (or vote “no” if not voted yet)
  – Commit iff all vote “yes”
• Transaction blocks if a memory node crashes
  – Must wait for memory node to recovery from disk
Sinfonia’s example usage: SinfoniaFS
General use of mini-transaction in SinfoniaFS

1. If local cache is empty, load it
2. Make modifications to local cache
3. Issue a mini-transaction to check the validity of cache, apply modification
4. If mini-transaction fails, reload cached item and try again
Mini-transaction usage example: append to file

- Find a free block in cached freemap
- Issue mini-transaction with
  - Compare items: cached inode, free status of the block
  - Write items: inode, append new block, freemap, new block
- If mini-transaction fails, reload cache
Summary:

• ACID transaction
  – Recovery relies on WAL logging
  – Concurrency control can use 2PL to achieve serializability

• Distributed transactions use 2PC for commit
  – 2PC is not fault tolerant