Atomic Commit and Concurrency Control

Let’s Scale Strong Consistency!

1. Atomic Commit
   • Two-phase commit (2PC)

2. Serializability
   • Strict serializability

3. Concurrency Control:
   • Two-phase locking (2PL)
   • Optimistic concurrency control (OCC)

Atomic Commit

• Atomic: All or nothing

• Either all participants do something (commit) or no participant does anything (abort)

• Common use: commit a transaction that updates data on different shards

Transaction Examples

• Bank account transfer
  • Turing → $100
  • Lovelace ↔ $100

• Maintaining symmetric relationships
  • Lovelace FriendOf Turing
  • Turing FriendOf Lovelace

• Order product
  • Charge customer card
  • Decrement stock
  • Ship stock
Relationship with Replication

- Replication (e.g., RAFT) is about doing the same thing multiple places to provide fault tolerance.
- Sharding is about doing different things multiple places for scalability.
- Atomic commit is about doing different things in different places together.

Focus on Sharding for Today

Atomic Commit

- Atomic: All or nothing.
- Either all participants do something (commit) or no participant does anything (abort).
- Atomic commit accomplished with two-phase commit protocol (2PC).
Two-Phase Commit

• Phase 1
  • Coordinator sends Prepare request to all participants
  • Each participant votes yes or no
    • Sends yes or no back to coordinator
    • Typically acquires locks if vote yes
  • Coordinator inspects all votes
    • If all yes, then commit
    • If any no, then abort

• Phase 2
  • Coordinator sends Commit or Abort to all participants
  • If commit, each participants acts
  • Each participant releases locks
  • Each participant sends Ack back to coordinator

Unilateral Abort

• Any participant can cause an abort

• With 100 participants, if 99 vote yes and 1 votes no => abort!

• Common reasons to abort:
  • Cannot acquire required lock
  • No memory or disk space available to do write
  • Transaction constraint fails, e.g., Alan does not have $100

• Q: Why do we want unilateral abort for atomic commit?

Atomic Commit

• All-or-nothing

• Unilateral abort

• Two-phase commit: Prepare -> Commit/abort

Lets Scale Strong Consistency!

1. Atomic Commit
   • Two-phase commit (2PC)

2. Serializability
   • Strict serializability

3. Concurrency Control:
   • Two-phase locking (2PL)
   • Optimistic concurrency control (OCC)
Two Concurrent Transactions

transaction sum(A, B):
begin_tx
a ← read(A)
b ← read(B)
print a + b
commit_tx

transaction transfer(A, B):
begin_tx
a ← read(A)
if a < 10 then abort_tx
else
write(A, a - 10)
write(B, b + 10)
commit_tx

Isolation Between Transactions

- **Isolation:** sum appears to happen either completely before or completely after transfer
  - i.e., appears that all ops of transaction happened together

- Schedule for transactions is an ordering of the operations performed by those transactions

Problem from Concurrent Execution

- Serial execution of transactions — transfer then sum:
  - transfer: r_A W_A r_B W_B ⊙
  - sum: r_A r_B ⊙

- Concurrent execution can result in state that differs from any serial execution:
  - transfer: deb_1 r_A W_A r_B W_B deb_2
  - sum: r_A r_B ⊙

  Time →

  ⊙ = commit

Isolation Between Transactions

- **Isolation:** sum appears to happen either completely before or completely after transfer
  - i.e., appears that all ops of transaction happened together

- Given a schedule of operations:
  - Is that schedule in some way “equivalent” to a serial execution of transactions?
Equivalence of Schedules

Two operations from different transactions are conflicting if:
1. They read and write to the same data item
2. The write and write to the same data item

Two schedules are equivalent if:
1. They contain the same transactions and operations
2. They order all conflicting operations of non-aborting transactions in the same way

Serializability

• A schedule is serializable if is equivalent to some serial schedule
  • i.e., non-conflicting ops can be reordered to get a serial schedule

A Serializable Schedule

• A schedule is serializable if is equivalent to some serial schedule
  • i.e., non-conflicting ops can be reordered to get a serial schedule

transfer: \( r_A \) \( w_A \) \( r_B \) \( w_B \) ©
sum: \( r_A \uparrow \) \( r_B \downarrow \)

Conflict-free!

Time \( \Rightarrow \) © = commit

A Non-Serializable Schedule

• A schedule is serializable if is equivalent to some serial schedule
  • i.e., non-conflicting ops can be reordered to get a serial schedule

transfer: \( r_A \) \( w_A \) \( r_B \) \( w_B \) ©
sum: \( r_A \) \( r_B \) ©

But in a serial schedule, sum's reads either both before \( w_A \) or both after \( w_B \)

Time \( \Rightarrow \) © = commit
Linearizability vs. Serializability

- **Linearizability**: guarantee about single ops on single objects
- Once write completes, all reads beginning later should reflect write

- **Serializability**: guarantee about transactions over ≥1 objects
- No real-time constraints imposed

- **Strict Serializability** = Serializability + real-time ordering
  - Intuitively Serializability + Linearizability
  - We’ll stick with only Strict Serializability for this class

Consistency Hierarchy

- Strict Serializability  e.g., Spanner
- Linearizability  e.g., RAFT, Paxos
- Sequential Consistency
- Causal+ Consistency  e.g., Bayou, COPS
- Eventual Consistency  e.g., Dynamo

Let's Scale Strong Consistency!

1. Atomic Commit
   - Two-phase commit (2PC)

2. Serializability
   - Strict serializability

3. Concurrency Control:
   - Two-phase locking (2PL)
   - Optimistic concurrency control (OCC)

Concurrency Control

- Concurrent execution can violate serializability

- Goal: **Control** that concurrent execution, so behave like single machine executing transactions one at a time

  => Concurrency control
Concurrency Control Strawman #1

- Big global lock
  - Acquire the lock when transaction starts
  - Release the lock when transaction ends
- Provides strict serializability
  - Just like executing transaction one by one because we are doing exactly that
- No concurrency at all
  - Terrible for performance: one transaction at a time

Locking

- Locks maintained on each shard
  - Transaction requests lock for a data item
  - Shard grants or denies lock
- Lock types
  - Shared: Need to have before read object
  - Exclusive: Need to have before write object

<table>
<thead>
<tr>
<th></th>
<th>Shared (S)</th>
<th>Exclusive (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared (S)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive (X)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Concurrency Control Strawman #2

- Grab locks independently, for each data item (e.g., bank accounts A and B)

  transfer: $A_rA_wA_b$ $B_rB_wB_b$

  sum: $A_rA_b A_rB_b B_rB_b$

  Permits this non-serializable interleaving

Two-Phase Locking (2PL)

- 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks
  - Growing phase: transaction acquires locks
  - Shrinking phase: transaction releases locks
- In practice:
  - Growing phase is the entire transaction
  - Shrinking phase is during commit
2PL Provide Strict Serializability

- 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks

transfer: $\triangle A r_A w_A \triangleright A \quad \triangle b r_b w_b \triangleleft B$

sum: $\triangle A r_A \triangle A r_A \triangle B r_b \triangle B w_b \triangleright B$

2PL precludes this non-serializable interleaving

Time $\to$

© = commit

$\triangle / \bigtriangledown = X- / S$-lock; $\triangleright / \triangleright = X- / S$-unlock

2PL and Transaction Concurrency

- 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks

transfer: $\triangle A r_A \quad \triangle A r_A \quad \triangle b r_b w_b \triangle B w_b \triangleright B$

sum: $\triangle A r_A \quad \triangle B r_b \triangleright B$

2PL permits this serializable, interleaved schedule

Time $\to$

© = commit

$\triangle / \bigtriangledown = X- / S$-lock; $\triangleright / \triangleright = X- / S$-unlock; $\bigstar = release\ all\ locks$

2PL Doesn’t Exploit All Opportunities for Concurrency

- 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks

transfer: $r_A w_A \quad r_b w_b \triangleright B$

sum: $r_A \quad r_b \triangleleft B$

2PL precludes this serializable, interleaved schedule

Time $\to$

© = commit

(locking not shown)

Issues with 2PL

- What do we do if a lock is unavailable?
  - Give up immediately?
  - Wait forever?

- Waiting for a lock can result in deadlock
  - Transfer has A locked, waiting on B
  - Sum has B locked, waiting on A

- Many different ways to detect and deal with deadlocks
More Concurrency Control Algorithms

- Optimistic Concurrency Control (OCC)
- Multi-Version Concurrency Control (MVCC)