Atomic Commit and Concurrency Control

COS 418/518: Distributed Systems
Lecture 16

Wyatt Lloyd
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)
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
Relationship with Replication

Replication Dimension

Sharding Dimension

A-F  A-F  A-F
G-L  G-L  G-L
M-R  M-R  M-R
S-Z  S-Z  S-Z
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 requests to all participants
  • Each participant votes yes or no
    • Sends yes or no vote back to coordinator
    • Typically acquires locks if they 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 participant does something
  • Each participant releases locks
  • Each participant sends an Ack back to the 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)
b ← read(B)
write(B, b+10)
commit_tx
Isolation Between Transactions

- **Isolation**: sum appears to happen either completely before or completely after transfer
  - i.e., it appears that all ops of a 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: \( r_A \ w_A \ r_B \ w_B \) ©
  sum: \( r_A \ r_B \) ©

Time → © = commit
Isolation Between Transactions

• **Isolation**: sum appears to happen either completely before or completely after **transfer**
  • i.e., it appears that all ops of a 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. They 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 it 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 it is equivalent to some serial schedule
  • i.e., non-conflicting ops can be reordered to get a serial schedule

**transfer:**  \( r_A \) \( w_A \)  \( r_B \) \( w_B \)  
\( \Rightarrow \)  
\( r_A \) \( r_B \)  
\( \Rightarrow \)  
\( \text{Serial schedule} \)  
\( \Rightarrow \)  
\( \text{Conflict-free!} \)  

\( \text{Time} \)  
\( \Rightarrow \)  
\( \text{©} = \text{commit} \)
A Non-Serializable Schedule

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

transfer: \( r_A \quad w_A \quad r_B \quad w_B \quad \copyright \)

sum: \( r_A \quad r_B \quad \copyright \)

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

© = commit
Linearizability vs. Serializability

- **Linearizability**: a guarantee about single operations on single objects
  - Once write completes, all reads that begin later should reflect that write

- **Serializability**: is guarantee about transactions over one or more objects
  - Doesn’t impose real-time constraints

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

Strict Serializability

→

Linearizability

→

Sequential Consistency

→

Causal+ Consistency

→

Eventual Consistency

e.g., Spanner

e.g., RAFT

e.g., Bayou

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

- We need to control that concurrent execution so we do things a single machine executing transactions one at a time would
  - 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)

\[
\begin{align*}
\text{transfer: } & A \xrightarrow{r_A} w_A \xrightarrow{\text{sum}} A \\
& B \xrightarrow{r_B} w_B \xrightarrow{\text{sum}} B
\end{align*}
\]

\[\text{Permits this non-serializable interleaving}\]

\[\text{Time } \rightarrow \quad \circ = \text{commit}\]

\[\blacktriangle / \blacktriangledown = \text{eXclusive-} / \text{Shared-lock}; \blacktriangleup / \blacktriangledown = \text{X-} / \text{S-unlock}\]
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: $\text{A} r_A w_A \text{A}$  $\text{B} r_B w_B \text{B} \textcopyright$

sum: $\text{A} r_A \text{A} \text{B} r_B \text{B} \textcopyright$

2PL precludes this non-serializable interleaving

Time $\rightarrow$

$\textcopyright$ = commit

$\text{A} / \text{B} = X- / S$-lock; $\text{A} / \text{B} = X- / S$-unlock
2PL and Transaction Concurrency

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

\[
\begin{align*}
\text{transfer:} & \quad \Delta_A r_A \quad \Delta_A w_A \quad \Delta_B r_B \quad \Delta_B w_B \ast \copyright \\
\text{sum:} & \quad \Delta_A r_A \quad \Delta_B r_B \ast \copyright
\end{align*}
\]

2PL permits this serializable, interleaved schedule

\[
\begin{align*}
\triangleright / \Delta &= X- / S\text{-lock}; & \quad \blacksquare / \Delta &= X- / S\text{-unlock}; & \quad \ast &= \text{release all locks}
\end{align*}
\]

Time \rightarrow

\copyright = \text{commit}
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

\[
\begin{align*}
\text{transfer:} & \quad r_A \, w_A \quad r_B \, w_B \, \copyright \\
\text{sum:} & \quad r_A \quad r_B \, \copyright
\end{align*}
\]

2PL precludes this serializable, interleaved schedule

Time \rightarrow

© = commit

(llocking 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)