I Can’t Believe It’s Not Causal!
Scalable Causal Consistency with No Slowdown Cascades

Syed Akbar Mehdi¹, Cody Littley¹, Natacha Crooks¹, Lorenzo Alvisi¹,⁴, Nathan Bronson², Wyatt Lloyd³

¹UT Austin, ²Facebook, ³USC, ⁴Cornell University
Causal Consistency: Great In Theory

- Lots of exciting research building scalable causal data-stores, e.g.,
  - COPS [SOSP 11]
  - Bolt-On [SIGMOD 13]
  - Chain Reaction [EuroSys 13]
  - Eiger [NSDI 13]
  - Orbe [SOCC 13]
  - GentileRain [SOCC 14]
  - Cure [ICDCS 16]
  - TARDiS [SIGMOD 16]
Causal Consistency: But In Practice ...

The middle child of consistency models

Reality: Largest web apps use eventual consistency, e.g.,

Espresso

TAO

Manhattan
Key Hurdle: Slowdown Cascades

Implicit Assumption of Current Causal Systems

Reality at Scale

Slowdown Cascade

Enforce Consistency

Wait
Replicated and sharded storage for a social network
Datacenter A

Read ($W_2$)

Datacenter B

Writes causally ordered as $W_1 \rightarrow W_2 \rightarrow W_3$
Current causal systems enforce consistency as a datastore invariant.
Alice’s advisor unnecessarily waits for Justin Bieber’s update despite not reading it
Slowdown cascades affect all previous causal systems because they enforce consistency inside the data store.

Alice’s advisor unnecessarily waits for Justin Bieber’s update despite not reading it.
Slowdown Cascades in Eiger (NSDI ’13)

Replicated write buffers grow arbitrarily because Eiger enforces consistency inside the datastore.
OCCULT

Observable Causal Consistency Using Lossy Timestamps
Observable Causal Consistency

Causal Consistency guarantees that each client observes a monotonically non-decreasing set of updates (including its own) in an order that respects potential causality between operations.

Key Idea:
Don’t implement a causally consistent data store
Let clients observe a causally consistent data store
How do clients observe a causally consistent datastore?
Writes accepted only by master shards and then replicated asynchronously in-order to slaves.
Each shard keeps track of a **shardstamp** which counts the writes it has applied.
Causal Timestamp: Vector of shardstamps which identifies a global state across all shards
Write Protocol: Causal timestamps stored with objects to propagate dependencies
Write Protocol: Server shardstamp is incremented and merged into causal timestamps
Read Protocol: Always safe to read from master
Read Protocol: Object’s causal timestamp merged into client’s causal timestamp
Read Protocol: Causal timestamp merging tracks causal ordering for writes following reads
Replication: Like eventual consistency; asynchronous, unordered, writes applied immediately
Replication: Slaves increment their shardstamps using causal timestamp of a replicated write.
Read Protocol: Clients do consistency check when reading from slaves.
b’s dependencies are delayed, but we can read it anyway!

Read Protocol: Clients do consistency check when reading from slaves
Read Protocol: Clients do consistency check when reading from slaves
Read Protocol: Resolving stale reads

Options:
1. Retry locally
2. Read from master
Causal Timestamp Compression

• What happens at scale when number of shards is (say) 100,000?

\[\text{Size(Causal Timestamp)} = 100,000?\]
Causal Timestamp Compression: Strawman

- To compress down to $n$, conflate shardstamps with same ids modulo $n$

![Compress]

- Problem: False Dependencies
- Solution:
  - Use system clock as the next value of shardstamp on a write
  - Decouples shardstamp value from number of writes on each shard
Causal Timestamp Compression: Strawman

- To compress down to $n$, conflate shardstamps with same ids modulo $n$

1000  89  13  209

Compress

1000  209

- Problem: Modulo arithmetic still conflates unrelated shardstamps
Causal Timestamp Compression

• **Insight:** Recent shardstamps more likely to create false dependencies
• Use high resolution for recent shardstamps and conflate the rest

<table>
<thead>
<tr>
<th>Shardstamps</th>
<th>4000</th>
<th>3989</th>
<th>3880</th>
<th>3873</th>
<th>3723</th>
<th>3678</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shard IDs</td>
<td>45</td>
<td>89</td>
<td>34</td>
<td>402</td>
<td>123</td>
<td>*</td>
</tr>
</tbody>
</table>

• 0.01 % false dependencies with just 4 shardstamps and 16K logical shards
Transactions in OCCULT

Scalable causally consistent general purpose transactions
Properties of Transactions

A. Atomicity
B. Read from a causally consistent snapshot
C. No concurrent conflicting writes
Properties of Transactions

A. *Observable* Atomicity

B. *Observably* Read from a causally consistent snapshot

C. No concurrent conflicting writes
Properties of Transactions

A. *Observable Atomicity*

B. *Observably* Read from a causally consistent snapshot

C. No concurrent conflicting writes

**Properties of Protocol**

1. No centralized timestamp authority (e.g. per-datacenter)
   - Transactions ordered using causal timestamps
2. Transaction commit latency is independent of number of replicas
Properties of Transactions

A. *Observable* Atomicity
B. *Observably* Read from a causally consistent snapshot
C. No concurrent conflicting writes

**Three Phase Protocol**

1. **Read Phase**
   - Buffer writes at client
2. **Validation Phase**
   - Client validates A, B and C using causal timestamps
3. **Commit Phase**
   - Buffered writes committed in an observably atomic way
Alice and her advisor are managing lists of students for three courses.
Observable atomicity and causally consistent snapshot reads enforced by single mechanism.
Start $T_1$
$r(a) = []$
$w(a = [Abe])$

Transaction $T_1$: Alice adding Abe to course $a$
Start $T_1$

\[ r(a) = [] \]

\[ w(a = [\text{Abe}]) \]

Commit $T_1$

Transaction $T_1$: After Commit
Transaction $T_2$: Alice moving Bob from course $b$ to course $c$
Atomicity through causality:
Make writes dependent on each other

Observable Atomicity: Make writes causally dependent on each other
Start $T_1$
- $r(a) = []$
- $w(a = [Abe])$
Commit $T_1$

Start $T_2$
- $r(b) = [Bob]$
- $r(c) = [Cal]$
- $w(b = [])$
- $w(c = [Bob, Cal])$
Commit $T_2$

**Observable Atomicity:** Same commit timestamp makes writes causally dependent on each other
Start $T_1$
- $r(a) = []$
- $w(a = \{Abe\})$
- Commit $T_1$

Start $T_2$
- $r(b) = \{Bob\}$
- $r(c) = \{Cal\}$
- $w(b = [])$
- $w(c = \{Bob, Cal\})$
- Commit $T_2$

Transaction writes replicate asynchronously
Alice’s advisor reads the lists in a transaction
Transactions maintain a Read Set to validate atomicity and read from causal snapshot.
Transactions maintain a Read Set to validate atomicity and read from causal snapshot.
Validation failure: \( c \) knows more writes from grey shard than applied at the time \( b \) was read.
Start $T_1$
- $r(a) = []$
- $w(a = [Abe])$
- Commit $T_1$

Start $T_2$
- $r(b) = [Bob]$
- $r(c) = [Cal]$
- $w(b = [])$
- $w(c = [Bob, Cal])$
- Commit $T_2$

Start $T_3$
- $r(b) = [Bob]$
- $r(c) = [Bob, Cal]$
- $r(a) = []$

**Ordering Violation:** Detected in the usual way. Red Shard is stale!
Properties of Transactions

A. *Observable* Atomicity

B. *Observably* Read from a causally consistent snapshot

C. No concurrent conflicting writes

Three Phase Protocol

1. Read Phase
   - Buffer writes at client

2. Validation Phase
   - Client validates
   - a. Validate Read Set to verify A and B
   - b. Validate Overwrite Set to verify C

3. Commit Phase
   - Buffered writes committed in an observably atomic way
Evaluation
Evaluation Setup

• Occult implemented by modifying Redis Cluster (baseline)
• Evaluated on CloudLab
  • Two datacenters in WI and SC
  • 20 server machines (4 server processes per machine)
  • 16K logical shards
• YCSB used as the benchmark
  • For graphs shown here read-heavy (95% reads) workload with zipfian distribution
• We show cost of providing consistency guarantees
Goodput Comparison

![Graph showing Goodput Comparison]

- Occult Transactions
- Occult Single-Key
- Redis Cluster

4 shardstamps per causal timestamp
Effect of slow nodes on Occult Latency

![Graph showing the effect of slow nodes on occult latency. The x-axis represents percentiles (50th, 75th, 90th, 95th, 99th), and the y-axis represents Log10 (Latency us). The graph compares different numbers of slow nodes: 0, 2, 4, and 6, with latency values for each percentile.]
Conclusions

• Enforcing causal consistency in the data store is vulnerable to slowdown cascades

• Sufficient to ensure that clients observe causal consistency:
  • Use lossy timestamps to provide the guarantee
  • Avoid slowdown cascades

• Observable enforcement can be extended to causally consistent transactions
  • Make writes causally dependent on each other to observe atomicity
  • Also avoids slowdown cascades