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]
  - GentleRain [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
Key Hurdle: Slowdown Cascades

Implicit Assumption of Current Causal Systems

Reality at Scale

Enforce Consistency

Wait

Wait

Slowdown Cascade
Replicated and sharded storage for a social network
Datacenter A

Writes causally ordered as $W_1 \rightarrow W_2 \rightarrow W_3$

Datacenter B
Current causal systems enforce consistency as a datastore invariant.
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
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 $n$

\[
\begin{array}{c}
1000 \quad 89 \quad 13 \quad 209 \\
\end{array}
\]

\[
\begin{array}{c}
1000 \quad 209 \\
\end{array}
\]

• Problem: False Dependencies
Causal Timestamp Compression: Strawman

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

\[
\begin{array}{cccc}
1000 & 89 & 13 & 209 \\
\end{array}
\]

• 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 from $N$ to $n$, conflate shardstamps with same ids modulo $n$

\[
\begin{array}{c}
1000 & 89 & 13 & 209 \\
\end{array}
\]

\[
\begin{array}{c}
1000 & 209 \\
\end{array}
\]

Compress

• Problem: Modulo arithmetic still confflates 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>

Catch-all shardstamp
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>Catch-all shardstamp</td>
<td>3678</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shard IDs</th>
<th>45</th>
<th>89</th>
<th>34</th>
<th>402</th>
<th>123</th>
<th>*</th>
</tr>
</thead>
</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 authorities (e.g. per-datacenter)
   - Transactions ordered using causal timestamps
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 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 same mechanism
Transaction $T_1$: Alice adding Abe to course $a$
Start $T_1$

$r(a) = []$

$w(a = [Abe])$

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
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 $T_2$

**Observable Atomicity:** Same commit timestamp makes writes causally dependent on each other
Transaction writes replicate asynchronously

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

1. Start T₁
   r(a) = []
   w(a = [Abe])
   Commit T₁

2. Start T₂
   r(b) = [Bob]
   r(c) = [Cal]
   w(b = [])
   w(c = [Bob, Cal])
   Commit T₂

Datacenter A

Datacenter B
Alice’s advisor reads the lists in a transaction
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 $T_2$

Alice’s advisor reads the lists in a transaction

Delayed!
Transactions maintain a Read Set to validate atomicity and causal snapshot reads.

- **Start T₁**: 
  - `r(a) = []` 
  - `w(a = [Abe])` 
  - Commit T₁ 

- **Start T₂**: 
  - `r(b) = [Bob]` 
  - `r(c) = [Cal]` 
  - `w(b = [])` 
  - `w(c = [Bob, Cal])` 
  - Commit T₂ 

- **Start T₃**: 
  - `r(b) = [Bob]` 

**Datacenter A** 
- **Master**: 
  - `a = [Abe]` 
  - `b = []` 
  - `c = [Bob, Cal]` 

**Slave**: 
- `a = []` 
- `b = [Bob]` 
- `c = [Bob, Cal]` 

**Datacenter B** 
- **Master**: 
  - `a = []` 
  - `b = []` 
  - `c = [Bob, Cal]` 

**Slave**: 
- `a = []` 
- `b = [Bob]` 
- `c = [Bob, Cal]`
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 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
Properties of Transactions

A. Observable Atomicity
B. Observably read from causally consistent snapshot
C. No concurrent conflicting writes

Three Phase Protocol

1. Read Phase
   - Buffer writes at client

2. Validation Phase
   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
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

**Diagram:**
- **Y-axis:** Goodput (million ops/s)
- **X-axis:** Num Ops per Transaction (T_{size})
- **Legend:**
  - Orange square: Occult Transactions
  - Black line: Occult Single-Key
  - Blue line: Redis Cluster
Goodput Comparison

<table>
<thead>
<tr>
<th>Num Ops per Transaction ($T_{size}$)</th>
<th>Occult Transactions</th>
<th>Occult Single-Key</th>
<th>Redis Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>8</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>12</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>14</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>16</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>18</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>20</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
</tr>
</tbody>
</table>

- Occult Transactions: 31% increase
- Occult Single-Key: 8.7% increase
- Redis Cluster: 39.6% increase

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

![Bar chart showing the effect of slow nodes on occult latency.](chart.png)

- **50th Percentile**: 280us
- **75th Percentile**: 390us
- **90th Percentile**: 800us
- **95th Percentile**: 1.6ms
- **99th Percentile**: 47.1ms

The chart compares latency for different percentiles with varying numbers of slow nodes.
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