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

Syed Akbar Mehdi, Cody Littley, and Natacha Crooks, *The University of Texas at Austin;*
Lorenzo Alvisi, *The University of Texas at Austin and Cornell University;*
Nathan Bronson, *Facebook;* Wyatt Lloyd, *University of Southern California*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

This paper is included in the Proceedings of the
14th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the
14th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by USENIX.

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

Syed Akbar Mehdi[1], Cody Littley[1], Natacha Crooks[1], Lorenzo Alvisi[1,4], Nathan Bronson[2], and Wyatt Lloyd[3]

[1]UT Austin, [2]Facebook, [3]USC, [4]Cornell University

Abstract

We describe the design, implementation, and evaluation of Occult (**O**bservable **C**ausal **C**onsistency **U**sing **L**ossy **T**imestamps), the first scalable, geo-replicated data store that provides causal consistency to its clients without exposing the system to the possibility of *slowdown cascades*, a key obstacle to the deployment of causal consistency at scale. Occult supports read/write transactions under PC-PSI, a variant of Parallel Snapshot Isolation that contributes to Occult's immunity to slowdown cascades by weakening how PSI replicates transactions committed at the same replica. While PSI insists that they all be totally ordered, PC-PSI simply requires total order Per Client session. Nonetheless, Occult guarantees that all transactions read from a causally consistent snapshot of the datastore without requiring any coordination in how transactions are asynchronously replicated.

1 Introduction

Causal consistency [7] appears to be ideally positioned to respond to the needs of the sharded and geographically replicated data stores that support today's large-scale web applications. Without imposing the high latency of stronger consistency guarantees [30, 38], it can address many issues that eventual consistency leaves unresolved. This brings clear benefits to users and developers: causal consistency is all that is needed to preserve operation ordering and give Alice assurance that Bob, whom she had defriended before posting her Spring-break photos, will not be able to access her pictures, even though Alice and Bob access the photo-sharing application using different replicas [13, 20, 39]. Yet, causal consistency has not seen widespread industry adoption.

This is not for lack of interest from the research community. In the last few years, we have learned that no guarantee stronger than real-time causal consistency can be provided in a replicated data store that combines high availability with convergence [43], and that, conversely, it is possible to build convergent causally-consistent data stores that can efficiently handle a large number of shards [10, 14, 27, 28, 39, 40].

We submit that industry's reluctance to deploy causal consistency is in part explained by the inability of its current implementations to comply with a basic commandment for scalability: do not let your performance be determined by your slowest component. In particular, current causal systems often prevent a shard in replica R from applying a write w until all shards in R have applied all the writes that causally precede w. Hence, a slow or failed shard (a common occurrence in any large-enough deployment) can negatively impact the entire system, delaying the visibility of updates across many shards and leading to growing queues of delayed updates. As we show in Section 2, these effects can easily snowball to produce the "slowdown cascades" that Facebook engineers recently indicated [8] as one of the key challenges in moving beyond eventual consistency.

This paper presents Occult (**O**bservable **C**ausal **C**onsistency **U**sing **L**ossy **T**imestamps), the first geo-replicated and sharded data store that provides causal consistency to its clients without exposing the system to slowdown cascades. To make this possible, Occult shifts the responsibility for the enforcement of causal consistency from the data store to its clients. The data store makes its updates available as soon as it receives them, and causal consistency is enforced on reads only for those updates that clients are actually interested in observing. In essence, Occult decouples the rate at which updates are applied from the performance of slow shards by optimistically *rethinking the sync* [48]: instead of enforcing causal consistency as an invariant of the data store, through its read-centric approach Occult appears to applications as *indistinguishable* from a system that does.

Because it never delays writes to enforce consistency, Occult is immune from the dangers of slowdown cascades. It may, however, delay read operations from shards that are lagging behind to ensure they appear consistent with what a user has already seen. We expect such delays to be rare in practice because a recent study of Facebook's eventually-consistent production system found that fewer than six out of every million reads were not causally consistent [42]. Our evaluation confirms this. We find that our prototype of Occult, when compared with the eventually-consistent system (Redis Cluster) it is derived from, increases the median latency by only 50μs, the 99th percentile latency by only 400μs for a read-heavy workload (4ms for a write-heavy workload), and reduces throughput by only 8.7% for a read-heavy workload (6.9% for a write-heavy workload).

Occult's read-centric approach, however, raises a thorny technical issue. Occult requires clients to determine how their

local state depends on the state of the *entire* data store; such global awareness is unnecessary in systems that implement causal consistency within the data store, where simply tracking the immediate predecessors of a write is enough to determine when the write should be applied [39]. In principle, it is easy to use vector clocks [29, 44] to track causal dependencies at the granularity of objects or shards. However, their overhead at the scale that Occult targets is prohibitive. Occult instead uses *causal timestamps* that, by synthesizing a variety of techniques for compressing dependency information, can achieve high accuracy (reads do not stall waiting for updates that they do not actually depend on) at low cost. We find that 24-byte timestamps suffice to achieve an accuracy of 99.6%; 8 more bytes give an accuracy of 99.96%.

Causal timestamps also play a central role in Occult's support for scalable read-write transactions. Transactions in Occult operate under a variant of Parallel Snapshot Isolation [55]. Occult ensures that all transactions always *observe* a consistent snapshot of the system, even though the datastore no longer evolves through a sequence of monotonically increasing consistent snapshots. It uses causal timestamps to not only track transaction ordering but also atomicity (by making writes of a transaction *causally dependent on each other*). This novel approach is key to the scalability of Occult's transactions and their immunity to slowdown cascades. The responsibility for commit is again shifted to the client, which uses causal timestamps to detect if a transaction has observed an inconsistent state due to an ordering or atomicity violation and, if so, aborts it. Committed writes instead propagate asynchronously to slaves, allowing the commit logic to scale independently of the number of slave replicas.

In summary, the contributions of this paper include:

- A novel and light-weight read-centric implementation of causal consistency. By shifting enforcement to the clients, it ensures that they never observe non-causal states, while placing no restrictions on the data store.
- A new transactional isolation level called Per-Client Parallel Snapshot Isolation (PC-PSI), a variant of PSI, that contributes to Occult's immunity to slowdown cascades by weakening how PSI replicates transactions committed at the same replica.
- A novel scalable protocol for providing PC-PSI that uses causal timestamps to enforce both atomicity and transaction ordering and whose commit latency is independent of the number of replicas in the system.
- An implementation and evaluation of Occult, the first causally consistent store that implements these ideas and is immune to slowdown cascades.

2 Motivation

Causal consistency ensures that clients observe their own updates and read from a state that includes all operations that they have previously observed. This promise aims for a sweet spot in the debate on the guarantees that a sharded and geo-replicated data store should offer. On the one hand, causal consistency maintains most of the performance edge of eventual consistency [60] over strong consistency, as all replicas are available for reads under network partitions [30, 38]. On the other hand, it minimizes the associated baggage of increased programmer complexity and user-visible anomalies. By ensuring that all clients see updates that may potentially be causally related [34] in the same order, causal consistency can, for example, address the race conditions that a VP of Engineering at Twitter in a 2013 tech talk called "the biggest problem for Twitter" [33]: when fanning out tweets from celebrities with huge followings, some feeds may receive reactions to the tweets before receiving the tweets themselves.

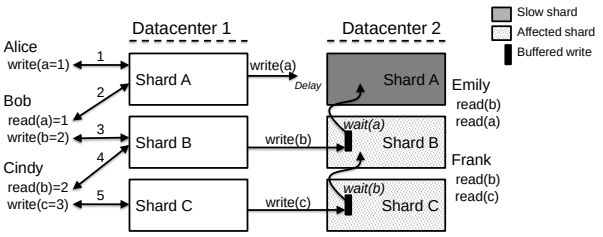


Figure 1: Example of a slowdown cascade in traditional causal consistency. Delayed replicated write(a) delays causally dependent replicated write(b) and write(c)

Despite these obvious benefits, however, causal consistency is largely not deployed in production systems, as existing implementations are liable to experience, at scale, one of the downsides of strong consistency: *slowdown cascades*.

2.1 Slowdown Cascades

When systems scale to sufficient size, failures become an inevitable and regular part of their operation [25, 26]. Performance anomalies, e.g., one node running with lower throughput than the rest of the system, are typical, and can be viewed as a kind of partial failure. Potential causes of such failures include abnormally-high read or write traffic, partially malfunctioning hardware, or a localized network issue, like congestion in a top-of-rack switch. In a partitioned system, a failure within a partition will inevitably affect the performance of that partition. A *slowdown cascade* occurs when the failure spills over to affect other partitions.

Industry has long identified the spectre of slowdown cascades as one of the leading reasons behind its reluctance to build strongly consistent systems [8, 17], pointing out how the slowdown of a single shard, compounded by *query amplification* (e.g., a single user request in Facebook can generate thousands of, possibly dependent, internal queries to many services), can quickly cascade to affect the entire system.

All existing causally consistent systems [14, 28, 39, 40, 63] are susceptible to slowdown cascades. The reason, in essence, is that, to present clients with a causally consistent data store, these systems delay applying a write w until after the data store reflects all the writes that causally precede w. For example, in Eiger [40] each replicated write w carries metadata that explicitly identifies the writes that directly precede w in the causal dependency graph. The datacenter

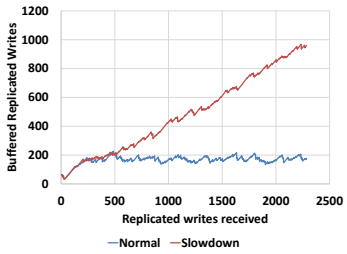


Figure 2: Average queue length of buffered replicated writes in Eiger under normal conditions and when a single shard is delayed by 100 ms.

then delays applying w until these dependencies have been applied locally. The visibility of a write within a shard can then become dependent on the timeliness of other shards in applying their own writes. As Figure 1 shows, this is a recipe for triggering slowdown cascades: because shard A of DC_2 lags behind in applying the write propagating from DC_1, all shards in DC_2 must also wait before they make their writes visible. Shard A's limping inevitably affects Emily's query, but also unnecessarily affects Frank's, which accesses exclusively shards B and C.

In practice, even a modest delay can trigger dangerous slowdown cascades. Figure 2 shows how a single slow shard affects the size of the queues kept by Eiger [40] to buffer replicated writes. Our setup is geo-replicated across two datacenters in Wisconsin and Utah, each running Eiger sharded across 10 physical machines. We run a workload consisting of 95% reads and 5% writes from 10 clients in Wisconsin and a read-only workload from 10 clients in Utah. We measure the average length of the queues buffering replicated writes in Utah. Larger queues mean that newer replicated writes take longer to be applied. If all shards proceed at approximately the same speed, the average queue length remains stable. However, if *any* shard cannot keep up with the arrival rate of replicated writes, then the average queue length across *all* shards grows indefinitely.

3 Observable Causal Consistency

To free causal consistency from slowdown cascades, we revisit what causal consistency *requires*.

Like every consistency guarantee, causal consistency defines a contract between the data store and its users that specifies, for a given set of updates, which values the data store is allowed to return in response to user queries. In particular, 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.

To abide by this contract, existing causally consistent data stores, when replicating writes, enforce internally a stronger invariant than the contract requires: they ensure that clients observe a monotonically non-decreasing set of updates by evolving their data store only through monotonically non-decreasing updates. This strengthening satisfies the contract but, as we saw in Section 2, leaves these systems vulnerable to slowdown cascades.

To resolve this issue, Occult moves the output commit to the clients: letting clients themselves determine when it is safe to read a value frees the data store to make writes visible to clients immediately, without having to first apply all causally preceding writes. Given the duties that many causally consistent data stores already place on their clients (such as maintaining the context of dependencies associated with each of the updates they produce [39, 40]), this is only a small step, but it is sufficient to make Occult impervious to slowdown cascades. Furthermore, Occult no longer needs its clients to be sticky (real-world systems like Facebook sometimes bounce clients between datacenters because of failures, load balancing, and/or load testing [8]). By empowering clients to determine independently whether a read operation is safe, it is no longer problematic to expose a client to the state of a new replica R2 that may not yet reflect some of the updates the client had previously observed on a replica R1.

The general outline of a system that moves the enforcement of causal consistency to read operations is straightforward. Each client c needs to maintain some metadata to encode the most recent state of the data store that it has observed. On reading an object o, c needs to determine whether the version of o that the data store currently holds is safe to read (i.e., if it reflects all the updates encoded in c's metadata): to this end, the data store could keep, together with o, metadata of its own to encode the most recent state known to the client that *created* that version of o. If the version is deemed safe to read, then c needs to update its metadata to reflect any new dependency; if it is not, then c needs to decide how to proceed (among its options: try again; contact a master replica guaranteed to have the latest version of o; or trade safety for availability by accepting a stale version of o).

The key challenge, however, is identifying an encoding of the metadata that minimizes both overhead and read latency. Since each object in the data store must be augmented with this metadata, the importance of reducing its size is obvious; keeping metadata small, however, reduces its ability to track causal dependencies accurately. Any such loss in definition is likely to introduce spurious dependencies between updates. Although these dependencies can never lead to slowdown cascades in Occult, they can increase the chances that read operations will be unnecessarily delayed. Occult's compressed causal timestamps leverage structural and temporal properties to strike a sweet spot between metadata overhead and accuracy (§5).

These causal timestamps have another, perhaps more surprising consequence: they allow Occult to offer the first scalable implementation of causal read-write transactions (§6). Just as the data-store need not be causal, transactions need not take effect atomically in the datastore. They simply need to *appear atomic* to clients. To achieve this, Occult makes

a transaction's writes *causally depend on each other*. This guarantees that clients that seek to read multiple writes from a transaction will independently determine that they must either observe all of the transactions's writes, or none. In contrast, transactions that seek to read a single of the transaction's writes will not be unnecessarily delayed until other replicas have applied writes that they are not interested in. Once again, this is a small step that yields big dividends: transactional writes need no longer be replicated synchronously for safety, obviating the possibility of slowdown cascades.

4 Occult: The Basic Framework

We first outline the system model and an idealized implementation of Occult's basic functionality: clients that read individual objects perceive the data store as causally consistent. We discuss how to make the protocol practical in §5 and sketch Occult's more advanced features (transactions) in §6.

4.1 System Model

Our system is a sharded and replicated key-value store where each replica is located in a separate datacenter with a full copy of the data. The keyspace is divided into a large number of *shards*, i.e., disjoint key ranges. There can be tens or hundreds of thousands of shards, of which multiple can be colocated on the same physical host.

We assume an asynchronous master-slave replication model, with a publicly designated master for every shard. This master shard accepts writes, and asynchronously, but in order, replicates writes to the slave shards. This design is common to several large-scale real-world systems [19, 20, 49, 51] that serve read-heavy workloads with online queries. Master-slave replication has higher write latency than multi-master schemes, but avoids the complexity of dealing with concurrent conflicting writes that can lead to lost updates [39] or require more complex programming models [24].

Clients in our system are co-located with a replica in the same datacenter. Each client reads from its local replica and writes to the master shard (possibly located in a remote replica); a client library enforces causal consistency for reads and attaches metadata to writes. While clients normally read from the shards in their replica, there is no requirement for them to be "sticky" (§3).

4.2 Causal Timestamps

Occult tracks and enforces causal consistency using shardstamps and causal timestamps. A shard's *shardstamp* counts the writes that the shard (master or slave) has accepted. A *causal timestamp* is a vector of shardstamps that identifies a global state across all shards: each entry stores the number of known writes from the corresponding shard. Keeping an entry per shard rather than per object trades-off accuracy against metadata overhead: in exchange for smaller timestamps, it potentially creates false dependencies among all updates to objects mapped to the same shard.

Occult uses causal timestamps for (*i*) encoding the most recent state of the data store observed by a client and (*ii*) capturing the set of causal dependencies for write operations. An object version o created by write w is associated with a causal timestamp that encodes all writes in w's *causal history* (i.e., w and all writes that causally preceded it). Upon reading o, a client updates its causal timestamp to the element-wise maximum of its current value and that of o's causal timestamp: the resulting vector defines the earliest state of the datastore that the client is now allowed to read from to respect causal consistency.

4.3 Basic Protocol

Causal consistency in Occult results from the cooperation between servers and client libraries enabled by causal timestamps. Client libraries use them to validate reads, update them after successful operations, and attach them to writes (Figure 3). Servers store them along with each object, and return one during reads. In addition, servers track the state of each shard using a dedicated shardstamp; when returned in response to a read request, it helps client libraries determine whether completing the read could potentially violate causal consistency (Figure 4).

Write Protocol Occult associates with any value written v a causal timestamp summarizing all of v's causal dependencies. The client library attaches its causal timestamp to every write and sends it to the master of the corresponding shard. The master increments the relevant shardstamp, updates the received causal timestamp accordingly, and stores it with the newly written value. It then asynchronously replicates the write to its slaves, before returning the shardstamp to the client library. Slaves receive writes from the master in order, along with the associated causal timestamps and shardstamps, and update their state accordingly. On receiving the shardstamp, the client library in turn updates *its* causal timestamp to reflect its current knowledge of the shard's state.

Read Protocol A client reads from its local server, which replies with the desired object's most recent value, that value's dependencies (i.e., its causal timestamp), and the current shardstamp of the appropriate shard. The returned shardstamp s makes checking for consistency straightforward. The client simply compares s with the entry of its own causal timestamp for the shard in question (call it s_c) . If s is at least s_c, then the shard already reflects all the local writes that the client has already observed.

When reading from the master shard, the consistency check is guaranteed to succeed. When reading from a slave, however, the check may fail: replication delays from the master shard in another datacenter may prevent a client from observing its own writes at the slave; or the client may have already observed a write in a different shard that *depends* on an update that has not yet reached the slave; .

If the check fails (i.e., the read is *stale*), the client has two choices. It can retry reading from the local replica

```
# cli_ts is the client's causal timestamp

def write(key, value):
  shrd_id = shard(key)
  master_server = master(shrd_id)
  shardstamp = master_server.write(key, value, cli_ts)
  cli_ts[shrd_id] = max(cli_ts[shrd_id], shardstamp)

def read(key):
  shrd_id = shard(key)
  local_server = local(shrd_id)
  value, deps, shardstamp = local_server.read(key)
  cli_ss = cli_ts[shrd_id]
  if isSlave(local_server) and shardstamp < cli_ss:
      return finishStaleRead(key)
  else: cli_ts = entrywise_max(cli_ts, deps)
      return value
```

Figure 3: Client Library Pseudocode

```
1  def write(key, value, deps): #(on masters)
2    shrd_id = shard(key)
3    shardstamps[shrd_id] += 1
4    shardstamp = shardstamps[shrd_id]
5    deps[shard_id] = shardstamp
6    store(key, value, deps)
7    for s in mySlaves(shard_id):
8      async(s.replicate(key, value, deps, shardstamp))
9    return shardstamp
10
11 def replicate(key, value, deps, shardstamp): #(on slaves)
12   shardstamps[shard(key)] = shardstamp
13   storeValue(key, value, deps)
14
15 def read(key):
16   shardstamp = shardstamps[shard(key)]
17   return (getValue(key), getDeps(key), shardstamp)
```

Figure 4: Server Pseudocode

until the shardstamp advances enough to clear the check. Alternatively, it can send the read to the master shard, which always reflects the most recent state of the shard, at the cost of increased latency and additional load on the master. Occult adopts a hybrid strategy: it retries locally for a maximum of r times (with an exponentially increasing delay between retries) and only then reads from the master replica. This approach resolves most stales quickly, while preventing clients from overloading their local slaves with excessive retries.

Finally, the client updates its causal timestamp to reflect the dependencies included in the causal timestamp returned by the server, ensuring that future successful reads will never be inconsistent with the last read value.

5 Causal Timestamp Compression

The above protocol relies on causal timestamps with an entry per shard, a prohibitive proposition when the number of shards N can be in the hundreds of thousands. Occult compresses their size to n entries (with $n \ll N$) without introducing many spurious dependencies.

A first attempt: structural compression Our most straightforward attempt—*structural compression*—maps all shards whose ids are congruent modulo n to the same entry, reducing a causal timestamps' size from N to n at the cost of generating spurious dependencies [58]. The impact of these dependencies on performance (in the form of delayed reads) worsens when shards have widely different shardstamps. Suppose shards i and j map to the same entry s_c and their shardstamps read, respectively, 100 and 1000. A client that writes to j will fail the consistency check when reading from a slave of i until i has received at least 1000 writes. In fact, if i never receives 1000 writes, the client will always failover to reading from i's master shard.

These concerns could be mitigated by requiring master shards to periodically advance their shardstamp and then replicate this advancement to their slaves, independent of the write rate from clients. However, fine-tuning the frequency and magnitude of this synchronization is difficult without explicit coordination between i and j. A better solution is instead to rely on *loosely synchronized shardstamps* based on real, rather than logical, clocks [6]. This guarantees that shardstamps differ by no more than the relative offset between their clocks, independent of the write rate on different master shards.

Finally, to reduce the impact of clock skew on creating false dependencies, the master for shard i can use the causal timestamp ts received from a client on a write operation to more tightly synchronize its shardstamp with those of other shards that the client has recently accessed. Rather than blindly using the current value cl of the physical clock of the server on which it is hosted, i can simply set its shardstamp to be larger than the maximum among (*i*) its current shardstamp; (*ii*) cl; and (*iii*) the highest of the values in ts.

Temporal compression Though using real clocks reduces the chances of generating spurious dependencies, it does not fully address the fundamental limitation of using modulo arithmetic to compress causal timestamps: it is still quite likely that shards with relatively far-apart shardstamps will be mapped to the same entry in the causal timestamp vector.

The next step in our refinement is guided by a simple intuition: recent shardstamps are more likely to generate spurious dependencies than older ones. Thus, rather than mapping a roughly equal number of shards to each of its n entries, *temporal compression* focuses a disproportionate fraction of its ability to accurately resolve dependencies on the shards with the most recent shardstamps. Adapting to our purposes a scheme first devised by Adya and Liskov [6], clients assign an individual entry in their causal timestamp to the $n-1$ shards with the most recent shardstamps they have observed. Each entry also explicitly stores the corresponding shard id. All other shards are mapped to the vector's "catch-all" last entry. One may reasonably fear that conflating all but $n-1$ shards in the same entry will lead, when a client tries to read from one of the conflated shards, to a large number of failed consistency checks—but it need not be so. For a large-enough n, the catch-all entry will naturally reflect updates that were accepted a while ago. Thus, when a client tries to read from a

conflated shard i, it is quite likely that the shardstamp of i will have already exceeded the value stored in the catch-all entry.

To allow causal timestamps to maintain the invariant of explicitly tracking the shards with the $n-1$ highest observed shardstamps, we must slightly revise the client's read and write protocols in Figure 3. The first change involves write operations on a shard currently mapped to the catch-all entry. When the client receives back that shard's current shardstamp, it compares it to those of the $n-1$ shards that its causal timestamp is currently tracking explicitly. The shard with the smallest shardstamp joins the ranks of the conflated and its shardstamp, if it exceeds the current value, becomes the new value of the catch-all entry for the conflated shards. The second change occurs on reads and concerns how the client's causal timestamp is merged with the one returned with the object being read. The shardstamps in either of the two causal timestamps are sorted, and only the shards corresponding to the highest $n-1$ shardstamps are explicitly tracked going forward; the others are conflated, and the new catch-all entry updated to reflect the new dependencies it now includes.

Isolating datacenters With either structural or temporal compression, the effectiveness of loosely synchronized timestamps in curbing spurious dependencies can be significantly affected by another factor: the interplay between the time it takes for updates to replicate across datacenters and the relative skew between the datacenters' clocks. Consider two datacenters, A and B, and assume for simplicity a causal timestamp consisting of a single shardstamp. Clocks within each datacenter are closely synchronized and we can ignore their skew. Say, however, that A's clocks run s ms ahead of those in B, that the average replication delay between datacenters is r ms, and that the average interval between consecutive writes at masters is i ms. Assume now that a client c in A writes to a local master node and updates its causal timestamp with the shardstamp it receives. If c then immediately tries to read from a local slave node, c's shardstamp will be ahead of the slave's by about $(s+r+i)$ ms: until the latter catches up, no value read from it will be deemed safe. For clients in B, meanwhile, the window of inconsistency under the same circumstances would be much shorter: just $(-s+r+i)$ ms, potentially leading to substantially fewer stale reads.

This effect can be significant (§8.2.1). The master write interval i, even with a read-heavy Zipfian workload, is less than 1 ms in our experiments. However, the replication delay r can range from a few tens to over 100 ms and cross datacenter clock skew s can be tens of milliseconds even when using NTP [3] (clock skew between nodes in the same datacenter is often within 0.5-2ms). Thus, if masters are distributed across datacenters, the percentage of stale reads experienced by clients of different datacenters can differ by orders of magnitude.

We solve this problem using distinct causal timestamps for each datacenter. On writes, clients use the returned shardstamp to update the causal timestamp of the datacenter hosting the relevant master shard. On reads, clients update each of their datacenter-specific causal timestamps using the corresponding causal timestamps returned by the server.

Two factors mitigate the additional overhead caused by datacenter-specific causal timestamps. First, the number of causal timestamps does not grow with the number of datacenters, but rather with the number of datacenters with master shards, which can be significantly lower [19]. Second, because clocks within each datacenter are closely synchronized, these causal timestamps need fewer entries to achieve a given target in the percentage of stale reads.

6 Transactions

Many applications can benefit from the ability to read and write multiple objects atomically. To this end, Occult builds on the system described for single-key operations to provide general-purpose read-write transactions. To the best of our knowledge, Occult is the first causal system to support general-purpose transactions while being scalable and resilient to slowdown cascades.

Transactions in Occult run under a new isolation property called Per-Client Snapshot Isolation (PC-PSI), a variant of Parallel Snapshot Isolation (PSI) [55]. PSI is an attractive starting point because it aims to strike a careful balance between the competing concerns of strong guarantees (important for developing applications) and scalable low-latency operations. On the one hand, PSI requires that transactions read from a causally consistent snapshot and precludes concurrent conflicting writes. On the other hand, PSI takes a substantial step towards improving scalability by letting transactions first commit at their local datacenter and subsequently replicate their effects asynchronously to other sites (while preserving causal ordering). In doing so, PSI sidesteps the requirement of a total order on *all* transactions, which is the primary scalability bottleneck of Snapshot Isolation [15] (a popular guarantee in non-distributed systems).

PSI's scalability, however, is ultimately undermined by the constraints its implementation imposes on the order in which transactions are to be replicated, leaving it unnecessarily vulnerable to slowdown cascades. Specifically, PSI totally orders all transactions that commit at a replica, and it requires this order to be respected when the transactions are replicated at other sites [55]. For instance, suppose the transactions in Figure 5 are executed by four different clients on the same replica. Under PSI, they would be totally ordered as $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$. If, when these transactions are applied at a different replica, any of the shards in charge of applying T_2 is slow, the replication of T_3 and T_4 will be delayed, even though neither has a read/write dependency on T_2.

PC-PSI removes these unnecessary constraints. Rather than totally ordering all transactions that were coincidentally located on the same replica, PC-PSI only requires transactions to be replicated in a way that respects both read/write

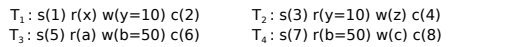


Figure 5: PSI requires transactions to be replicated in commit order. $s(i)$ and $c(j)$ mean respectively start (commit) at timestamp i (j).

dependencies and the order of transactions that belong to the same client session (even when the client is not sticky). This is sufficient to ensure semantically relevant dependencies, i.e., if Alice defriends Bob in one transaction and then later posts her Spring-break photos in another transaction, then Bob will not be able to view her photos, regardless of which replica he reads from. At the same time, it allows Occult to support transactions while minimizing its vulnerability to slowdown cascades.

Like PSI, PC-PSI precludes concurrent conflicting writes. When implementing read-write transactions, this guarantee is crucial to removing the danger of anomalies like lost updates [15]. When writes are accepted at all replicas, as in most existing causally consistent systems [10, 27, 28, 39, 40] this guarantee comes at the cost of expensive synchronization [35], crippling scalability and driving up latency. Not so in Occult, whose master-slave architecture makes it straightforward and inexpensive to enforce, laying the basis for Occult's low-latency read/write transactions.

6.1 PC-PSI Specification

To specify PC-PSI, we start from PSI. In particular, we leverage recent work [23] that proves PSI is equivalent to *lazy consistency* [6]. This isolation level is known [5] to be the weakest to simultaneously provide two guarantees at the core of PC-PSI: (*i*) transactions observe a consistent snapshot of the database and (*ii*) write-write conflicts are not allowed. We thus build on the theoretical framework behind the specification of lazy consistency [5], adding to it the requirement that transactions in the same client session must be totally ordered.

Concretely, we associate with the execution H of a set of transactions a directed serialization graph $DSG(H)$, whose nodes consist of committed transactions and whose edges mark the conflicts (*rw* for read-write, *ww* for write-write, *wr* for write-read) that occur between them. To these, we add a fourth set of edges: $T_i \xrightarrow{sd} T_j$ if some client c first commits T_i and then T_j (*sd* is short for *session dependency*).

The specification of PC-PSI then constrains the set of valid serialization graphs. In particular, a valid $DSG(H)$ must not exhibit any of the following anomalies:

Aborted Reads A committed transaction T_2 reads some object modified by an aborted transaction T_1.

Intermediate Reads A committed transaction T_2 reads a version of an object x written by another transaction T_1 that was not T_1's final modification of x.

Circular Information Flow $DSG(H)$ contains a cycle consisting entirely of *wr*, *ww* and *sd* edges.

Missed Effects $DSG(H)$ contains a cycle that includes exactly one *rw* edge.

Intuitively, preventing Circular Information Flow ensures that if T_1 and T_2 commit and T_1 depends on T_2, then T_2 cannot depend on T_1. In turn, disallowing cycles with a single *rw* edge ensures that no committed transaction ever misses writes of another committed transaction on which it otherwise depends, i.e., committed transactions read from a consistent snapshot and write-write conflicts are prevented (§6.3).

6.2 Executing Read/Write Transactions

Occult supports read/write transactions via a three-phase optimistic concurrency protocol that, in line with the system's ethos, makes clients responsible for running the logic needed to enforce PC-PSI (see Appendix A for the protocol's pseudocode). First, in the *read phase*, a client c executing transaction T obtains from the appropriate shards the objects that T reads, and locally buffers T's writes. Then, in the *validation phase*, c ensures that all read objects belong to a consistent snapshot of the system that reflects the effects of all transactions that causally precede T. Finally, in the *commit phase*, c writes back atomically all objects updated by T.

Read phase For each object o read by T, c contacts the local server for the corresponding shard s_o, making sure, if the server is a slave, not to be reading a stale version (§4.3) of o—i.e., a version of o that is older than what c's causal timestamp already reflects about the state of s_o. If the read is successful, c adds o, its causal timestamp, and s_o's shardstamp to T's *read set*. Otherwise, after a tunable number of further attempts, c proceeds to read o from its master server, whose version is never stale. Meanwhile, all writes are buffered in T's *write set*. They are atomically committed to servers in the final phase. Thus only committed objects are read in this phase and cascading aborts are not possible.

Validation phase Validation involves three steps. In the first, c verifies that the objects in its read set belong to a consistent snapshot Σ_{rs}. It does so by checking that all pairs o_i and o_j of such objects are *pairwise consistent* [12], i.e., that the saved shardstamp of the shard s_{o_i} from which o_i was read is at least as up to date as the entry for s_{o_i} in the causal timestamp of o_j (and vice versa). If the check fails, T aborts.

In the second step, c attempts to lock every object o updated by a write w in T's write set by contacting the corresponding shard s_o on the master server. If c succeeds, Occult's master-slave design ensures that c has exclusive write access to the latest version of o (reads are always allowed); if not, c restarts this step of the validation phase until it succeeds (or possibly aborts T after n failed attempts). In response to a successful lock request, the master server returns two data items: 1) o's causal timestamp, and 2) the new shardstamp that will be assigned to w. c stores this information in T's *overwrite set*. Note that, since they have been obtained from the corresponding master servers, the causal timestamps of the objects in the overwrite set are guaranteed to be pairwise consistent, and therefore to define a consistent snapshot Σ_{ow}: Σ_{ow} captures the updates of all transactions that T would depend on after committing.

To ensure that T is not missing any of these updates, in the final step of validation c checks that Σ_{rs} is at least as recent as Σ_{ow}. If the check fails, T aborts.

Commit phase c computes T's *commit timestamp* ts_T by first initializing it to the causal timestamp of the snapshot Σ_{rs} from which T read, and by then updating it to account for the shardstamps, saved in T's overwrite set, assigned to T's writes. The value of $ts_T[i]$ is thus set to the largest between (*i*) the highest value of the i-th entry of any of the causal timestamps in T's read set, and (*ii*) the highest shardstamp assigned to any of the writes in T's write set that update an object stored on a shard mapped to entry i. c then writes back the objects in T's write set to the appropriate master server, with ts_T as their causal timestamp. Finally, to ensure that any future transaction executed by this client will be (causally) ordered after T, c sets its own causal timestamp to ts_T.

The commit phase enforces a property that is crucial for Occult's scalability: it guarantees that transactions are atomic even though Occult replicates their writes asynchronously. Because the commit timestamp ts_T both reflects all writes that T performs and is used as the causal timestamp of every object that T updates, ts_T makes all of these updates, in effect, causally dependent on one another. As a result, any transaction whose read set includes any object o in T's write set will necessarily either become dependent on all the updates that T performed, or none of them.

6.3 Correctness

To implement PC-PSI, the protocol must prevent Aborted Reads, Intermediate Reads, Circular Information Flow, and Missed Effects. The optimistic nature of the protocol trivially yields the first two conditions, as writes are buffered locally and only written back when transactions commit. Occult also precludes Circular Information Flow. Since clients acquire write locks on all objects before modifying them, transactions that modify the same objects cannot commit concurrently and interleave their writes (no *ww* cycles). Cycles consisting only of *ww*, *wr*, and *sd* edges are instead prevented by the structure of OCC, whose read phase strictly precedes all writes: if a sequence of *ww*/*wr*/*sd* edges leads from T_1 to T_2, then T_1 must have committed before T_2, and could not have observed the effects of T_2 or created a write with a lower causal timestamp than T_2's.

Finally, Occult's validation phase prevents Missed Effects. By contradiction, suppose that all transactions involved in a DSG cycle with a single anti-dependency (*rw*) edge have passed the validation phase. Let T be the transaction from which that edge originates, ending in T^*. Let T_{-1} immediately precede T in the cycle. Let o be the object written by T^* whose update T missed. Either T_{-1} and T^* are one, or T_{-1} *wr*/*ww*/*sd* depends on T^*: either way, Occult's protocol ensures that the commit timestamp of T_{-1} is at least as large as that of T^*. By assumption, T missed some update to o: hence, the shardstamp for o's shard s_o in T's readset must be smaller that the corresponding entry in the commit timestamps of T^* and T_{-1}. There are three cases:

(*i*) $T_{-1} \xrightarrow{sd} T$. The client that issued both T_{-1} and T must have decreased its causal timestamp after committing T_{-1}, but the protocol ensures causal timestamps increase monotonically.

(*ii*) $T_{-1} \xrightarrow{ww} T$. Since T overwrites an object updated by T_{-1}, T's overwrite set must include T_{-1}'s commit timestamp. But then T would fail in validating its read set against its overwrite set, since the latter has a larger entry corresponding to s_o than the former.

(*iii*) $T_{-1} \xrightarrow{wr} T$. Since T reads an object updated by T_{-1}, its read set contains T_{-1}'s commit timestamp. But then T would fail in validating its read set, since the object updated by T_{-1} and the version of o read by T would be pairwise inconsistent.

Each case leads to a contradiction: hence no such cycle can occur and no effects are missed.

7 Fault Tolerance

Server failures Slave failures in Occult only increase read latency as slaves never accept writes and read requests to failed slaves eventually time-out and redirect to the master. Master failures are more critical. First, as in all single-master systems [56], no writes can be processed on a shard with a failed master. Second, in common with all asynchronously replicated systems [11, 14, 39, 40, 56], Occult exhibits a vulnerability window during which writes executed at the master may not yet have been replicated to slaves and may be lost if the master crashes. These missing writes may cause subsequent client requests to fail: if a client c's write to object o is lost, c cannot read o without violating causality. This scenario is common to all causal systems for which clients do not share fate with the servers to which they write. Occult's client-centric approach to causal consistency, however, creates another dangerous scenario: as datacenters are not themselves causally consistent, writes can be replicated out of order. A write y that is dependent on a write x can be replicated to another datacenter despite the loss of x, preventing any subsequent client from reading both x and y.

Master failures can be handled using well-known techniques: individual machine failures within a datacenter can be handled by replicating the master locally using chain-replication [59] or Paxos [36], before replicating asynchronously to other replicas.

Client failures A client failure for single-key operations impacts only the failed client as neither reads nor writes create temporary server state. In transactional mode, however, clients modify server state during the commit phase: they acquire locks on objects in the transaction's write-set and write back new values. A client failure during the transaction commit process may thus cause locks to be held indefinitely by failed clients, preventing other transactions from committing. Such failures can be handled by augmenting Occult with Bernstein's cooperative termination protocol [16] for coor-

dinator recovery [32, 64]. Upon detecting a suspected client failure, individual shards can attempt to elect themselves as backup coordinator (using an instance of Paxos to ensure that a single coordinator is elected). The backup coordinator can then appropriately terminate the transaction (by committing it if a replica shard successfully received an unlock request with the appropriate transaction timestamp using the buffered writes at every replica, or aborting it otherwise).

8 Evaluation

Our evaluation answers three questions:

1. How well does Occult perform in terms of throughput, latency, and transaction abort rate?
2. What is its overhead when compared to an eventually-consistent system?
3. What is the effect of server slowdowns on Occult?

We have implemented Occult by modifying Redis Cluster [4], the distributed implementation of the widely-used Redis key-value store. Redis Cluster divides the entire key-space into N logical shards (default N = 16K), which are then evenly distributed across the available physical servers. Our causal timestamps track shardstamps at the granularity of logical shards to avoid dependencies on the physical location of the data.

For a fair comparison with Occult, we modify our Redis Cluster baseline to allow reads from slaves (Redis Cluster by default uses primary-backup [53] replication for fault tolerance). We further modify the Redis client [2] to, like Occult, allow for client locality: the client prioritizes reading from shards in its local datacenter and executes write operations at the master shard.

8.1 Experimental Setup

Unless otherwise stated, we run our experiments on CloudLab [1, 52] with 20 server and 20 client machines evenly divided across two datacenters in Wisconsin (WI) and South Carolina (SC); the cross-datacenter ping latency is 39ms. Each machine has dual Intel E5-2660 10-core CPUs and dual-port Intel 10Gbe NICs, with respectively 160GB memory (WI) and 256GB (SC). Our experiments use public IP addresses, routable between CloudLab sites, which are limited to 1Gbps. Each server machine runs four instances of the server process, with each server process being responsible for $N/40$ logical shards. Half of all shards have a master in WI and a slave in SC; the other half have the opposite configuration.

Client machines run the Yahoo! Cloud Serving Benchmark (YCSB) [21]. We run experiments with both of YCSB's Zipfian and Uniform workloads but, for brevity, show results only for the Zipfian distribution, more representative of real workloads. Prior to the experiments, we load the cluster with 10 million records following YCSB's default, i.e., keys varying in size up to 23B and 1KB values. We report results at peak goodput, running for at least 100 seconds and then excluding 10-second ramp-up and ramp-down periods. Goodput measures successful operations per second, e.g., a read that needs to be retried four times will only be counted once towards goodput. The bottleneck resource for all experiments is out bound network bandwidth on the hottest master. The CPU on the hottest master is nearly saturated ($>$ 90% utilization) and would almost immediately bottleneck each system at a similar throughput if we were able to increase the network bandwidth.

8.2 Performance and Overhead

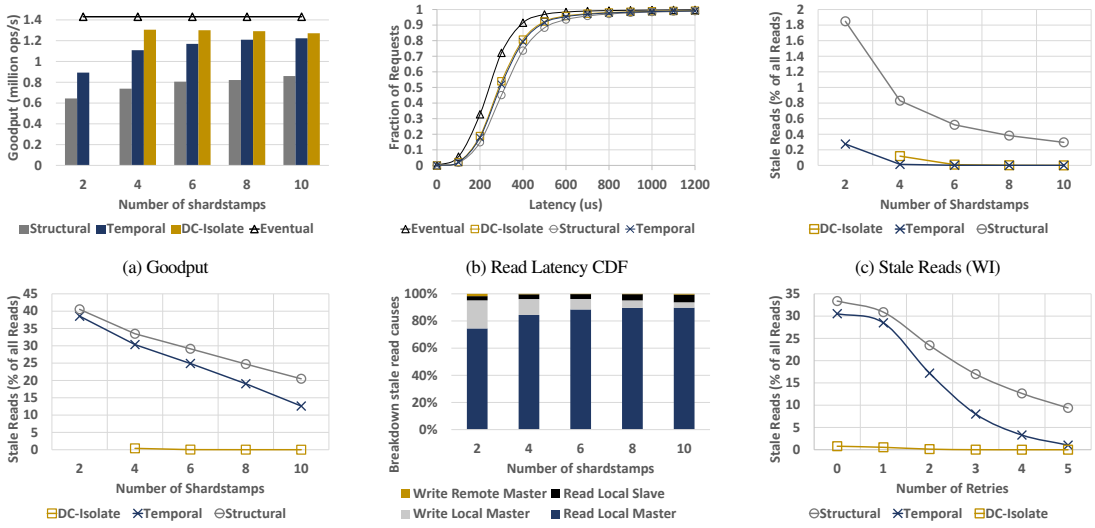
8.2.1 Single Key Operations

We first quantify the overhead of enforcing causal consistency in Occult. We show results for a read-heavy (95% reads, 5% writes) workload, which is more interesting and challenging for our system. Write-heavy workloads performed better in general: we include them in Appendix B.1 for completeness.

We compare system throughput as a function of causal timestamp size, for each of the previously described schemes (structural, temporal, and temporal with datacenter isolation), with Redis cluster as the baseline. Temporal compression requires a minimum of two entries per causal timestamp; adding datacenter isolation (DC-Isolate), doubles this number, so that the smallest number of shardstamps used by DC-Isolate is four.

In the best case (DC-Isolate scheme with four-entry timestamps), Occult's performance is competitive with Redis, despite providing much stronger guarantees: its goodput is only 8.7% lower than Redis (Figure 6a) and its mean and tail latency are, respectively, only 50 µs and 400 µs higher than in Redis (Figure 6b). Other schemes perform either systematically worse (Structural), or require twice the number of shardstamps to achieve comparable performance (Temporal). The low performance of the structural and temporal schemes are due to their high stale read rate (Figures 6c and 6d). In contrast, DC-Isolate has very a low percentage of stale reads even with small causal timestamps. Its slight drop in goodput is primarily due to Occult's other source of overhead: the CPU, network, and storage cost of attaching and storing timestamps to requests and objects. These results highlight the tension between overhead and precision: larger causal timestamps reduce the amount of stale reads (as evidenced by the improved performance of the temporal scheme when vector size grows), but worsen overhead (the goodput of the DC-Isolate scheme actually drops slightly as the number of shardstamps increases).

Achieving a low stale read rate with few shardstamps, as DC-Isolate does, is thus crucial to achieving good performance. Key to its success is its ability to track timestamps from different datacenters independently. Consider Figures 6c and 6d: in these experiments we simply count the percentage of stale reads but do not retry locally or read from the remote master. Observe that the temporal and structural schemes suffer from a significantly higher stale read rate in



(a) Goodput (b) Read Latency CDF (c) Stale Reads (WI)

(d) Stale Reads (SC) (e) Analysis of Stale Read Causes (f) Retrying Stale reads

Figure 6: Measurement and analysis of Occult's overhead for single key operations. Spatial, Temporal or DC-Isolate mean that we run Occult using those compression methods while Eventual indicates our baseline, i.e., Redis Cluster. WI means Wisconsin datacenter and SC means South Carolina datacenter.

the SC datacenter. To understand why, we instrumented the code to track metadata related to the last operation to modify a client's causal timestamp before it does a stale read. We discovered that almost 96% of stale reads occur when the client writes or reads from a local master node immediately before reading from a local slave node (Figure 6e). If the local master node runs ahead (for instance, the SC datacenter has a positive offset of about 22 ms, as measured via *ntpdate*), the temporal scheme will declare all reads to the local slave as stale. In contrast, by tracking dependencies on a per-datacenter basis, DC-Isolate side-steps this issue, producing a low stale rate across datacenters.

8.2.2 Transactions

To evaluate transactions, we modify the workload generator of the YCSB benchmark to issue *start* and *commit* operations in addition to reads and writes. Operations are dispatched serially, i.e., operation i must complete before operation $i+1$ is issued. The resulting long duration of transactions are worst case scenario for Occult. The generator is parameterized with the required number of operations per transaction (T_{size}). We use the DC-Isolate scheme for Occult in all these experiments.

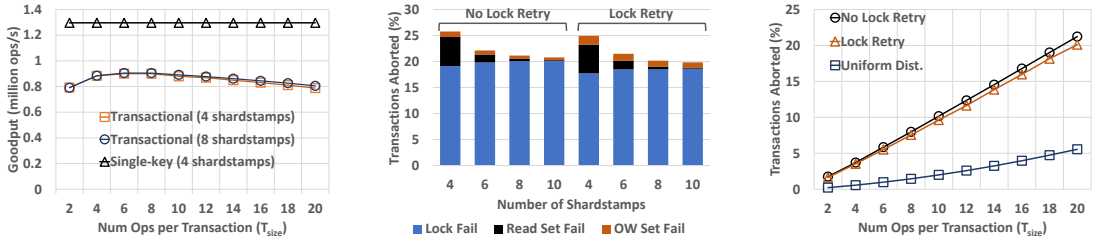
We show results for increasing values of T_{size}. For smaller values, most transactions in the workload are read-only, and as T_{size} increases most transactions become read-write . As Figure 7a shows, the overall goodput remains within 2/3 of the goodput of non transactional Occult (varying from 60% to 70%), even as T_{size} increases and aborts become more likely. Figures 7b and 7c analyze the causes of these aborts. Recall from §6.2 that aborts can occur because of either (*i*) validation failures of the read/overwrite sets or (*ii*) failure to acquire locks on keys being written. Figure 7b fixes $T_{size}=20$ and classifies aborts into these three categories. We find that aborts are dominated by the failure to acquire locks. Furthermore, due to the highly skewed nature of the YCSB zipfian workload, >80% of these lock-fail aborts are due to contention on the 50 hottest keys. This high contention also explains the limited benefit of retrying to acquire locks. Figure 7b also shows that increasing the number of shardstamps almost completely eliminates aborts due to failed validations of the read set and roughly halves aborts due to failed validations of the overwrite set. Finally, in Figure 7c retrying lock acquisition has slightly better impact at larger values of T_{size} when most transactions are read-write. For comparison, we show the abort rate on a uniform distribution.

8.2.3 Resource Overhead

To quantify the resource overhead of Occult over Redis Cluster, we measure the CPU usage (using *getrusage*()) and the total bytes sent and received over 120 secs for both systems at the same throughput (1.27Mop/s) and report the average of five runs, averaged over the 80 server processes.

Overall CPU usage increases by 7% with a slightly higher increase on slaves (8%) than masters (6%). This difference is due to stale read retries in Occult. Output bandwidth increases by 8.8%, while input bandwidth increases by 49%, as attaching metadata to read requests with a key size of at most 23B has a much larger impact than attaching it to replies carrying 1KB values.

Finally, we measure storage overhead by loading both Redis and Occult with 10 million records and measuring the increase in memory usage of each server process. Storing four shardstamps with each key results in an increase, on average, of 3% for Occult over Redis. Storing 10 shardstamps instead results in an increase of 4.9%.



(a) Goodput

(b) Abort Rate as $f(shardstamps)$ ($T_{size}=20$)

(c) Abort Rate as $f(T_{size})$ ($shardstamps=8$)

Figure 7: Transactions in Occult

8.3 Impact of slow nodes

Occult is by design immune to the slowdown of a server cascading to affect the entire system. Nonetheless, the slowdown of a server does introduce additional overhead relative to an eventually-consistent system. In particular, slowing down slaves increases the stale rate, which in turn increases retries on that slave and remote reads from its corresponding master. We measure these effects by artificially slowing down the replication of writes at a number of slave nodes, symmetrically increasing their number from one to three per datacenter—two to six overall. This causes a slowdown of around 2.5% to 7.5% of all nodes. We notice that, at peak throughput, the node containing the hottest key serves around 3× more operations than the nodes serving keys in the tail of the distribution. We evaluate slowdowns of the tail nodes separately from the hot nodes, which we slowdown in decreasing order of load, starting from the hottest node.

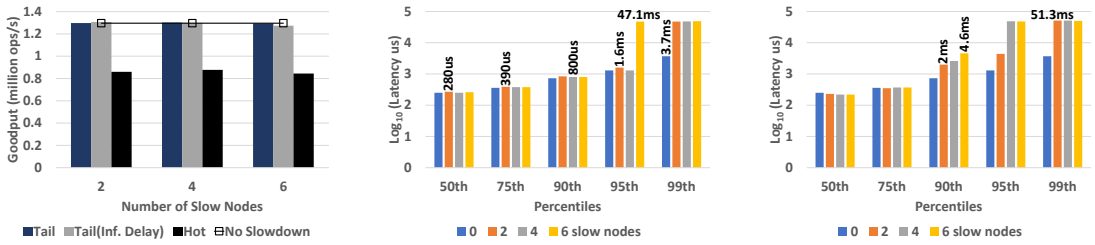
We first delay replicated writes on tail nodes by 100 ms, which, as Figure 8a shows, does not affect throughput: even at peak throughput for the cluster, only the hottest nodes are actually CPU or network saturated. As such, tail nodes (master or slave) still have spare capacity. When clients failover to the master (after n local retries), this spare capacity absorbs the additional load. In contrast, read latency is affected (Fig 8b). Though median, 75th, and 90th percentile latencies remain unchanged because reads to non-slow nodes are unaffected by the presence of slow servers, tail latencies increase significantly as the likelihood of hitting a lagging server and reading a stale value increases. Thus, increasing slow nodes from two to six first makes the 99th percentile and then the 95th percentile latency jump to around 48ms. This includes $n=4$ local retries by the client (after delays of 0, 1, 2, and 4 ms) and finally contacting the master in a remote datacenter (39 ms away). Having a large delay of 100 ms and $n=4$ means that our experiment actually evaluates an arbitrarily large slowdown, since almost all client reads to slow slaves eventually fail over to the master. We confirm this by setting the delay to infinite: the results for both throughput (Figure 8a) and latency (not shown) are identical to the 100 ms case.

Slowing down the hot nodes impacts both throughput and latency. The YCSB workload we use completely saturates the hottest master and its slave. Unlike in the previous experiments, the hot master does not have any spare capacity to handle failovers, and throughput suffers (Figure 8a). Slowing more than two slave nodes does not decrease throughput further becuase their respective masters have spare capacity. Figure 8c shows that, as expected given the skewed workload, slowing down an increasing number of hot nodes increases the 99th and 95th percentile latencies faster than slowing down tail nodes (Figure 8b). The median and 75th percentile latencies remain unchanged as before.

9 Related Work

Scalable Causal Consistency COPS [39] tracks causal consistency with explicit dependencies and then enforces it pessimistically by checking these dependencies before applying remote writes at a replica. COPS strives to limit the loss of throughput caused by the metadata and messages needed to check dependencies by exploiting transitivity. ChainReaction [10], Orbe [27], and GentleRain [28] show how to reduce these dependencies further by using Bloom filters, dependency matrices, and a single timestamp, respectively. These techniques reduce metadata by making it more coarse-grained, which actually exacerbates slowdown cascades. Eiger [40] builds on COPS with a more general data model, write-only transactions, and an improved read-only transaction algorithm. BoltOn [14] shows how to use shim layers to add pessimistic causal consistency to an existing eventually consistent storage system. COPS-SNOW [41] provides a new latency-optimal read-only transaction algorithm. Occult improves on this line of research by identifying the problem of slowdown cascades and showing how an optimistic approach to causal consistency can overcome them. In addition, all of these systems provide weaker forms of transactions than Occult: Eiger provides read-only and write-only transactions, while all other systems provide only read-only transactions or no transactions at all.

Pileus [56] and Tuba [11] (which adds reconfigurability to Pileus) provide a range of consistency models that clients can dynamically choose between by specifying an SLA that assigns utilities to different combinations of consistency and latency. Pileus has several design choices that are similar to Occult: it uses a single master, applies writes at replicas without delay (i.e., is optimistic), uses a timestamp to determine



(a) Goodput (b) Latency(slow tail nodes) (c) Latency(slow hot nodes)

Figure 8: Effect on overall goodput and read latency due to slow nodes in Occult

if a read value meets a given consistency level (including causal consistency), and can issue reads across different datacenters to meet a given consistency level. However, Pileus is not scalable as it uses a single logical timestamp as the client's state (which we show in our evaluation has a very high false positive stale rate) and evaluates with only a single node per replica. We consider an interesting avenue of future work to see if we can combine the focus of Pileus (consistency choice and SLAs) with Occult.

Cure [9] is a causally consistent storage system that provides read-write transactions. Cure is pessimistic and uses a single timestamp per replica to track and enforce causal dependencies. Cure provides a restricted form of read-write transactions that requires all operations to be on convergent and commutative replicated data types (CRDTs) [54]. Using CRDTs allows Cure to avoid coordination for writes and instead eventually merges conflicting writes, including those issued as part of read-write transactions. Occult, in contrast, is an optimistic system that provides read-write transactions for the normal data types that programmers are familiar with. Saturn [18], like Occult, tries to strike a balance between metadata overhead and false sharing by relying on "small labels" (like Cure) while selecting serializations at datacenters that minimize spurious dependencies.

Read/Write Transactions Many recent research systems with read/write transactions are limited to a single datacenter (e.g., [37, 46, 61, 62]) whereas most production systems are geo-replicated. Some geo-replicated research systems cannot scale to large clusters because they have a single point of serialization per datacenter [24, 55] while others are limited to transactions with known read and write sets [47, 57, 65].

Scalable geo-replicated transactional systems include Spanner [22], MDCC [32], and TAPIR [64]. Spanner is a production system at Google that uses synchronized clocks to reduce coordination for strictly serializable transactions. MDCC uses Generalized Paxos [36] to reduce wide-area commit latency. TAPIR avoids coordination in both replication and concurrency control to be able to sometimes commit a transaction in a single wide-area round trip. All of these systems provide strict serializability, a much stronger consistency level than what Occult provides. As a result, they require heavier-weight mechanisms for deciding to abort or commit transactions and will abort more often.

Rethinking the Output Commit Step We were inspired to rethink the output commit step for causal consistency by a number of previous systems: Rethink the Sync [48], which did it for local file I/O; Blizzard [45], which did it for cloud storage; Zyzzyva [31], which did it for Byzantine fault tolerance; and Speculative Paxos [50], which did it for Paxos.

10 Conclusion

This paper identifies slowdown cascades as a fundamental limitation of enforcing causal consistency as a global property of the datastore. Occult instead moves this responsibility to the client: the data store makes its updates available as soon as it receives them. Clients then enforce causal consistency on reads only for updates that they are actually interested in observing, using compressed timestamps to track causality. Occult follows the same philosophy for its scalable general-purpose transaction protocol: by ensuring that transactions read from a consistent snapshot and using timestamps to guarantee atomicity, it guarantees the strong properties of PSI while avoiding its scalability bottleneck.

11 Acknowledgements

We are grateful to our shepherd Jay Lorch for his dedication to making this paper as good as it could be, and to the anonymous reviewers for their insightful comments. This work would simply not have been possible without the patience and support of the amazing CloudLab team [1] throughout our experimental evaluation. This work was supported by the National Science Foundation under grant number CNS-1409555 and by a Google Faculty Research Award. Natacha Crooks was partially supported by a Google Doctoral Fellowship in Distributed Computing.

References

[1] CloudLab. https://www.cloudlab.us/.

[2] Jedis. https://github.com/xetorthio/jedis.

[3] Network Time Protocol. https://www.eecis.udel.edu/~mills/ntp.html.

[4] Redis Cluster Specification. http://redis.io/topics/cluster-spec.

[5] ADYA, A. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, 1999.

[6] ADYA, A., AND LISKOV, B. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing* (Santa Barbara, California, USA, 1997), PODC '97, ACM, pp. 73–82.

[7] AHAMAD, M., NEIGER, G., BURNS, J., KOHLI, P., AND HUTTO, P. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing 9*, 1 (1995), 37–49.

[8] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland, 2015), HOTOS'15, USENIX Association.

[9] AKKOORATH, D. D., TOMSIC, A. Z., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIA, N., AND SHAPIRO, M. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (June 2016), pp. 405–414.

[10] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 85–98.

[11] ARDEKANI, M. S., AND TERRY, D. B. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), OSDI '14, USENIX Association, pp. 367–381.

[12] BABAOĞLU, O., AND MARZULLO, K. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 55–96.

[13] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (San Jose, California, 2012), SoCC '12, ACM, pp. 22:1–22:7.

[14] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-On Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 761–772.

[15] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA, 1995), SIGMOD '95, ACM, pp. 1–10.

[16] BERNSTEIN, P., AND NEWCOMER, E. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., 1997.

[17] BIRMAN, K., CHOCKLER, G., AND VAN RENESSE, R. Toward a Cloud Computing Research Agenda. *SIGACT News 40*, 2 (June 2009), 68–80.

[18] BRAVO, M., RODRIGUES, L., AND VAN ROY, P. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the 12th ACM European Conference on Computer Systems* (2017), EuroSys '17, ACM.

[19] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, 2013), USENIX ATC'13, USENIX Association, pp. 49–60.

[20] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008), 1277–1288.

[21] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, 2010), SoCC '10, ACM, pp. 143–154.

[22] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[23] CROOKS, N., PU, Y., ALVISI, L., AND CLEMENT, A. Seeing is Believing: A Unified Model for Consistency and Isolation via States. *CoRR abs/1609.06670* (2016).

[24] CROOKS, N., PU, Y., ESTRADA, N., GUPTA, T., ALVISI, L., AND CLEMENT, A. TARDiS: A Branch-and-Merge Approach to Weak Consistency. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, 2016), SIGMOD '16, ACM, pp. 1615–1628.

[25] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[26] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 137–149.

[27] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th ACM Symposium on Cloud Computing* (Santa Clara, California, 2013), SOCC '13, ACM, pp. 11:1–11:14.

[28] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the 5th ACM Symposium on Cloud Computing* (2014), SOCC '14, ACM.

[29] FIDGE, C. J. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)* (February 1988), pp. 56–66.

[30] GILBERT, S., AND LYNCH, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News 33*, 2 (June 2002), 51–59.

[31] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems 27*, 4 (Jan. 2010), 7:1–7:39.

[32] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 113–126.

[33] KRIKORIAN, R. Twitter Timelines at Scale (video link. consistency discussion at 26m). http://www.infoq.com/presentations/Twitter-Timeline-Scalability, 2013.

[34] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[35] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[36] LAMPORT, L. Generalized Consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2004.

[37] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 71–86.

[38] LIPTON, R. J., AND SANDBERG, J. PRAM: A Scalable Shared Memory. Tech. Rep. TR-180-88, Princeton University, Department of Computer Science, August 1988.

[39] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 401–416.

[40] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 313–328.

[41] LU, H., HODSDON, C., NGO, K., MU, S., AND LLOYD, W. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association, pp. 135–150.

[42] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 295–310.

[43] MAHAJAN, P., ALVISI, L., AND DAHLIN, M. Consistency, Availability, and Convergence. Tech. Rep. UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin, 2011.

[44] MATTERN, F. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms* (1989), North-Holland/Elsevier, pp. 215–226.

[45] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association, pp. 257–273.

[46] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI'14, USENIX Association, pp. 479–494.

[47] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association.

[48] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. *ACM Transactions on Computer Systems 26*, 3 (Sept. 2008), 6:1–6:26.

[49] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 385–398.

[50] PORTS, D. R., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), USENIX Association, pp. 43–57.

[51] QIAO, L., SURLAKER, K., DAS, S., QUIGGLE, T., SCHULMAN, B., GHOSH, B., CURTIS, A., SEELIGER, O., ZHANG, Z., AURADAR, A., BEAVER, C., BRANDT, G., GANDHI, M., GOPALAKRISHNA, K., IP, W., JGADISH, S., LU, S., PACHEV, A., RAMESH, A., SEBASTIAN, A., SHANBHAG, R., SUBRAMANIAM, S., SUN, Y., TOPIWALA, S., TRAN, C., WESTERMAN, J., AND ZHANG, D. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 1135–1146.

[52] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[53] SCHNEIDER, F. B. Replication Management Using the State-Machine Approach. In *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 169–197.

[54] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Tech. Rep. HAL Id: inria-00555588, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[55] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 385–400.

[56] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania, 2013), SOSP '13, ACM, pp. 309–324.

[57] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 1–12.

[58] TORRES-ROJAS, F. J., AND AHAMAD, M. Plausible clocks: Constant size logical clocks for distributed systems. *Distributed Computing 12*, 4 (Sept. 1999), 179–195.

[59] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 91–104.

[60] VOGELS, W. Eventually Consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[61] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 87–104.

[62] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 279–294.

[63] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada, 2015), Middleware '15, ACM, pp. 75–87.

[64] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 263–278.

[65] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania, 2013), SOSP '13, ACM, pp. 276–291.

A Pseudocode for Transactions

Listing 1: Interface of a Causal Timestamp

```
class CausalTimestamp:
  def init(N):
    V = [0] * N

# Get shardstamp for shard_id
  def getSS(shard_id):
    return V[shard_id]

# Return the shardstamp with maximum value
  def maxSS():
    return max(V)

# Update the shardstamp for shard_id to new_ss
  def updateSS(shard_id,new_ss):
    V[shard_id] = max(V[shard_id], new_ss)

# Merge another CausalTimestamp into this object
 def mergeCTS(other_cts):
    for i in range(0, len(V)):
      V[i] = max(V[i], other_cts[i])
```

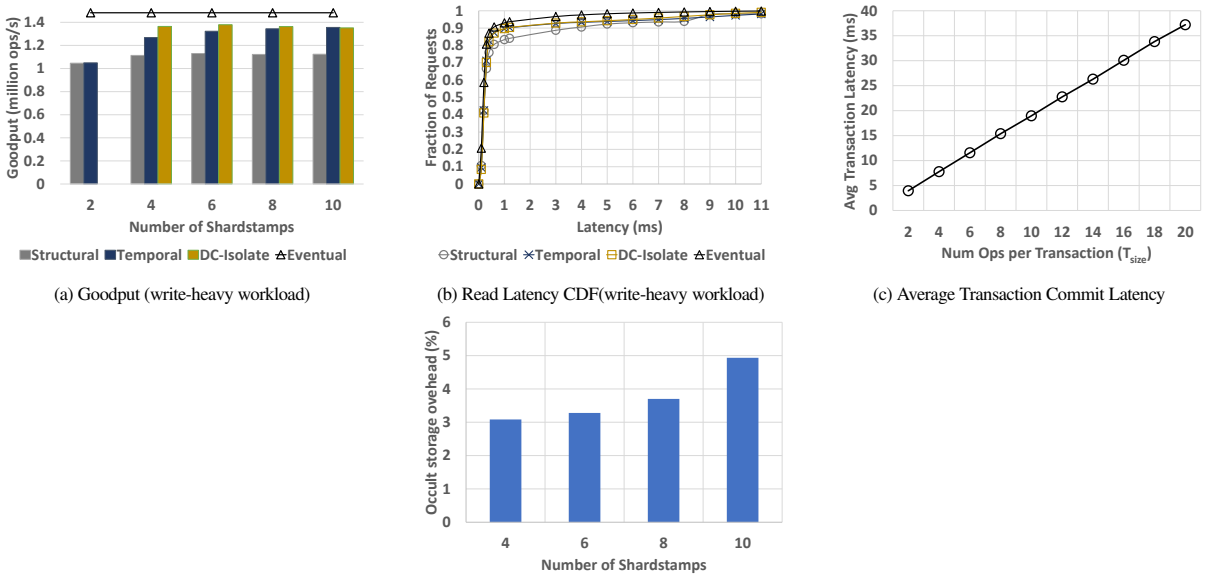
Listing 2: Server-side pseudocode

```
# allocate new shardstamp using loosely synchronized
# clocks as described in Section 5
def newShardstamp(max_cli_ss, shard_id):
  new_ss = max(currentSysTime(), max_cli_ss)
  if new_ss < shardstamps[shard_id]:
    return shardstamps[shard_id] + 1
  else:
    return new_ss + 1

def read(key):
  shardstamp = shardstamps[shard(key)]
  return (getValue(key),getDeps(key),shardstamp)

def prepare(tid, key, value, max_cli_ss):
  if not islocked(shard(key)):
    lockwrites(shard(key))
    if tid not in prepKV: # prepared txns key vals
      prepKV[tid] = list()
    prepKV[tid].append((key, value))
    shardstamp = shardstamps[shard(key)]
    new_ss = newShardstamp(max_cli_ss, shard(key))
    return (new_ss, getDeps(key))
  else:
    throw LOCKED

def commit_server(tid, deps):
  for key, value in prepKV[tid]:
    shardstamps[shard(key)] = deps.maxSS()
    store(key, value, deps)
    shardstamp = shardstamps[shard(key)])
    unlockwrites(shard(key))
    for s in mySlaves():
      async(s.replicate(key,value,deps,shardstamp)

def abort_server(tid):
  for key, value in prepKV[tid]:
    unlockwrites(key)
```

Note that if multiple transactions concurrently update different objects in the same shard *s*, in the commit phase each write *w* is applied at *s* (and at its slaves) in the (total) order determined by the value of the shardstamp assigned to *w* during the validation phase. The pseudocode achieves this property by locking shards instead of objects during the validation phase (§6.2).

Listing 3: Client-side pseudocode

```
# cli_ts is the client's causal timestamp
def startTransaction():
  TID = newTransactionID()
  ReadSet = set()
  OWSet = set() # Overwrite Set
  Writes = dict() # Writes done by this transaction
  cli_ts_save = copy(cli_ts)

def write(key, value):
  Writes[key] = value

def read(key):
  if key in Writes:
    return Writes[key] # Return the value we wrote
  else:
    shard_id = shard(key)
    local_server = local(shard_id)
    cli_ss = cli_ts.getSS(shard_id)
    value,deps,shardstamp = local_server.read(key)
    if isSlave(local_server) and shardstamp < cli_ss:
      value,deps,shardstamp = finishStaleRead(key)

    ReadSet.add(Elem(key,shard_id,deps,shardstamp))
    cli_ts.mergeCTS(deps)
    return value

def validate(S1, S2):
  for x in S1:
    for y in S2:
      if x.shardstamp < y.deps.getSS(x.shard_id):
        return False
  return True

def abortTransaction(prepared_servers, tid):
  cli_ts = cli_ts_save
  for server in prepared_servers:
    server.abort_server(tid)
  return False

def commitTransaction():
  prepared_servers = set()
  if not validate(ReadSet, ReadSet):
    return abortTransaction(prepared_servers, TID)

  for key, value in Writes:
    master_server = master(shard(key))
    try:
      max_ss = cli_ts.maxSS()
      new_ss,deps =
          master_server.prepare(TID,key,value,max_ss)
      cli_ts.updateSS(shard(key), new_ss)
      OWSet.add(Elem(key,shard(key),deps))
      prepared_servers.add(master_server)
    except LOCKED: #can retry lock here before abort
      return abortTransaction(prepared_servers, TID)

  if not validate(ReadSet, OWSet):
    return abortTransaction(prepared_servers, TID)
  else:
    for server in prepared_servers:
      server.commit_server(TID, cli_ts)
    return True
```



(a) Goodput (write-heavy workload)

(b) Read Latency CDF(write-heavy workload)

(c) Average Transaction Commit Latency

(d) Storage overhead of Occult over Redis Cluster

Figure 9: Miscellaneous additional evaluation. Spatial, Temporal or DC-Isolate mean that we run Occult using those compression methods while Eventual indicates our baseline, i.e., Redis Cluster.

B Additional Evaluation

In this section we show additional evaluation that could not be shown in the main paper due to space constraints. The experimental setup for this section is identical to the setup from section 8.1 of the main paper.

B.1 Performance on a write-heavy workload

Figures 9a and 9b show evaluation of non-transactional Occult on a write-heavy workload (75% reads, 25% writes) with a Zipfian distribution of operations. Overall Occult suffers less goodput overhead (6.9%) over Redis on this workload than the read-heavy workload. The median latency increase over Redis is still 50µs but tail latency increases by 4ms.

B.2 Commit Latency of Transactions

Figure 9c shows the average commit latency of transactions from *start* to *commit* as a function of T_{size}, i.e., the number of operations in each transaction. The linear rise in latency is because operations in our workload are dispatched serially as discussed in the evaluation of transactions in §8.2.

B.3 Storage overhead of Occult over Redis Cluster

Figure 9d shows the storage overhead of Occult over Redis Cluster with increasing number of shardstamps per causal timestamp. For this experiment, we loaded either system with 10 million records and measured the increase in memory usage of the server processes in Occult. The increase happens since Occult stores causal timestamps with each key.