Stronger Semantics for Low-Latency Geo-Replicated Storage

Wyatt Lloyd*, Michael J. Freedman*, Michael Kaminsky†, and David G. Andersen‡
*Princeton University, †Intel Labs, ‡Carnegie Mellon University

Abstract

We present the first scalable, geo-replicated storage system that guarantees low latency, offers a rich data model, and provides "stronger" semantics. Namely, all client requests are satisfied in the local datacenter in which they arise; the system efficiently supports useful data model abstractions such as column families and counter columns; and clients can access data in a causally-consistent fashion with read-only and write-only transactional support, even for keys spread across many servers.

The primary contributions of this work are enabling scalable causal consistency for the complex column-family data model, as well as novel, non-blocking algorithms for both read-only and write-only transactions. Our evaluation shows that our system, Eiger, achieves low latency (single-ms), has throughput competitive with eventually-consistent and non-transactional Cassandra (less than 7% overhead for one of Facebook's real-world workloads), and scales out to large clusters almost linearly (averaging 96% increases up to 128 server clusters).

1 Introduction

Large-scale data stores are a critical infrastructure component of many Internet services. In this paper, we address the problem of building a geo-replicated data store targeted at applications that demand fast response times. Such applications are now common: Amazon, EBay, and Google all claim that a slight increase in user-perceived latency translates into concrete revenue loss [25, 26, 41, 50].

Providing *low latency* to the end-user requires two properties from the underlying storage system. First, storage nodes must be near the user to avoid long-distance round trip times; thus, data must be replicated geographically to handle users from diverse locations. Second, the storage layer itself must be fast: client reads and writes must be local to that nearby datacenter and not traverse the wide area. Geo-replicated storage also provides the important benefits of availability and fault tolerance.

Beyond low latency, many services benefit from a *rich data model*. Key-value storage—perhaps the simplest data model provided by data stores—is used by a number of services today [4, 29]. The simplicity of this data model, however, makes building a number of interesting services overly arduous, particularly compared to the column-family data models offered by systems like BigTable [19] and Cassandra [37]. These rich data models provide hierarchical sorted column-families and numerical counters. Column-families are well-matched to services such as Facebook, while counter columns are particularly useful for numerical statistics, as used by collaborative filtering (Digg, Reddit), likes (Facebook), or re-tweets (Twitter).

Unfortunately, to our knowledge, no existing geo-replicated data store provides guaranteed low latency, a rich column-family data model, and *stronger consistency semantics*: consistency guarantees stronger than the weakest choice—eventual consistency—and support for atomic updates and transactions. This paper presents Eiger, a system that achieves all three properties.

The consistency model Eiger provides is tempered by impossibility results: the strongest forms of consistency—such as linearizability, sequential, and serializability—are impossible to achieve with low latency [8, 42] (that is, latency less than the network delay between datacenters). Yet, some forms of stronger-than-eventual consistency are still possible and useful, e.g., *causal consistency* [2], and they can benefit system developers and users. In addition, *read-only* and *write-only transactions* that execute a batch of read or write operations at the same logical time can strengthen the semantics provided to a programmer.

Many previous systems satisfy two of our three design goals. Traditional databases, as well as the more recent Walter [52], MDCC [35], Megastore [9], and some Cassandra configurations, provide stronger semantics and a rich data model, but cannot guarantee low latency. Redis [48], CouchDB [23], and other Cassandra configurations provide low latency and a rich data model, but not stronger semantics. Our prior work on COPS [43] supports low latency, some stronger semantics—causal consistency and read-only transactions—but not a richer data model or write-only transactions (see §7.8 and §8 for a detailed comparison).

A key challenge of this work is to meet these three goals while *scaling* to a large numbers of nodes in a

single datacenter, which acts as a single logical replica. Traditional solutions in this space [10, 12, 36], such as Bayou [44], assume a single node per replica and rely on techniques such as log exchange to provide consistency. Log exchange, however, requires serialization through a single node, which does not scale to multi-node replicas.

This paper presents Eiger, a scalable geo-replicated data store that achieves our three goals. Like COPS, Eiger tracks dependencies to ensure consistency; instead of COPS' dependencies on versions of keys, however, Eiger tracks dependencies on operations. Yet, its mechanisms do not simply harken back to the transaction logs common to databases. Unlike those logs, Eiger's operations may depend on those executed on other nodes, and an operation may correspond to a transaction that involves keys stored on different nodes.

Eiger's read-only and write-only transaction algorithms each represent an advance in the state-of-the-art. COPS introduced a read-only transaction algorithm that normally completes in one round of local reads, and two rounds in the worst case. Eiger's read-only transaction algorithm has the same properties, but achieves them using logical time instead of explicit dependencies. Not storing explicit dependencies not only improves Eiger's efficiency, it allows Eiger to tolerate long partitions between datacenters, while COPS may suffer a metadata explosion that can degrade availability.

Eiger's write-only transaction algorithm can atomically update multiple columns of multiple keys spread across multiple servers in a datacenter (i.e., they are atomic within a datacenter, but not globally). It was designed to coexist with Eiger's read-only transactions, so that both can guarantee low-latency by (1) remaining in the local datacenter, (2) taking a small and bounded number of local messages to complete, and (3) never blocking on any other operation. In addition, both transaction algorithms are general in that they can be applied to systems with stronger consistency, e.g., linearizability [33].

The contributions of this paper are as follows:

- The design of a low-latency, causally-consistent data store based on a column-family data model, including all the intricacies necessary to offer abstractions such as column families and counter columns.
- A novel non-blocking read-only transaction algorithm that is both performant and partition tolerant.
- A novel write-only transaction algorithm that atomically writes a set of keys, is lock-free (low latency), and does not block concurrent read transactions.
- An evaluation that shows Eiger has performance competitive to eventually-consistent Cassandra.

2 Background

This section reviews background information related to Eiger: web service architectures, the column-family data model, and causal consistency.

2.1 Web Service Architecture

Eiger targets large geo-replicated web services. These services run in multiple datacenters world-wide, where each datacenter stores a full replica of the data. For example, Facebook stores all user profiles, comments, friends lists, and likes at each of its datacenters [27]. Users connect to a nearby datacenter, and applications strive to handle requests entirely within that datacenter.

Inside the datacenter, client requests are served by a front-end web server. Front-ends serve requests by reading and writing data to and from storage tier nodes. Writes are asynchronously replicated to storage tiers in other datacenters to keep the replicas loosely up-to-date.

In order to scale, the storage cluster in each datacenter is typically partitioned across 10s to 1000s of machines. As a primitive example, Machine 1 might store and serve user profiles for people whose names start with 'A', Server 2 for 'B', and so on.

As a storage system, Eiger's *clients* are the front-end web servers that issue read and write operations on behalf of the human users. When we say, "a client writes a value," we mean that an application running on a web or application server writes into the storage system.

2.2 Column-Family Data Model

Eiger uses the column-family data model, which provides a rich structure that allows programmers to naturally express complex data and then efficiently query it. This data model was pioneered by Google's BigTable [19]. It is now available in the open-source Cassandra system [37], which is used by many large web services including EBay, Netflix, and Reddit.

Our implementation of Eiger is built upon Cassandra and so our description adheres to its specific data model where it and BigTable differ. Our description of the data model and API are simplified, when possible, for clarity.

Basic Data Model. The column-family data model is a "map of maps of maps" of named columns. The first-level map associates a key with a set of named column families. The second level of maps associates the column family with a set composed exclusively of either columns or super columns. If present, the third and final level of maps associates each super column with a set of columns. This model is illustrated in Figure 1: "Associations" are a column family, "Likes" are a super column, and "NSDI" is a column.

bool	⟵	batch_mutate	({key→mutation})
bool	⟵	atomic_mutate	({key→mutation})
{key→columns}	⟵	multiget_slice	({key, column_parent, slice_predicate})

Table 1: Core API functions in Eiger's column family data model. Eiger introduces `atomic_mutate` and converts `multiget_slice` into a read-only transaction. All calls also have an `actor_id`.

	User Data		Associations				
			Friends			Likes	
	ID	Town	Alice	Bob	Carol	NSDI	SOSP
Alice	1337	NYC	-	3/2/11	9/2/12	9/1/12	-
Bob	2664	LA	3/2/11	-	-	-	-
⋮							

Figure 1: An example use of the column-family data model for a social network setting.

Within a column family, each *location* is represented as a compound key and a single value, i.e., "Alice:Assocs:Friends:Bob" with value "3/2/11". These pairs are stored in a simple ordered key-value store. All data for a single row must reside on the same server.

Clients use the API shown in Table 1. Clients can insert, update, or delete columns for multiple keys with a `batch_mutate` or an `atomic_mutate` operation; each mutation is either an `insert` or a `delete`. If a column exists, an `insert` updates the value. Mutations in a `batch_mutate` appear independently, while mutations in an `atomic_mutate` appear as a single atomic group.

Similarly, clients can read many columns for multiple keys with the `multiget_slice` operation. The client provides a list of tuples, each involving a key, a column family name and optionally a super column name, and a slice predicate. The slice predicate can be a `(start,stop,count)` three-tuple, which matches the first `count` columns with names between `start` and `stop`. Names may be any comparable type, e.g., strings or integers. Alternatively, the predicate can also be a list of column names. In either case, a *slice* is a subset of the stored columns for a given key.

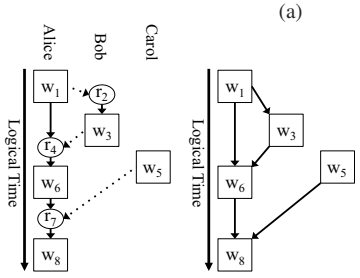
Given the example data model in Figure 1 for a social network, the following function calls show three typical API calls: updating Alice's hometown when she moves, ending Alice and Bob's friendship, and retrieving up to 10 of Alice's friends with names starting with B to Z.

```
batch_mutate(Alice→insert(UserData:Town=Rome))

atomic_mutate(Alice→delete(Assocs:Friends:Bob),
              Bob→delete(Assocs:Friends:Alice))

multiget_slice({Alice, Assocs:Friends, (B, Z, 10)})
```

Counter Columns. Standard columns are updated by `insert` operations that overwrite the old value. Counter columns, in contrast, can be commutatively updated using an `add` operation. They are useful for maintaining numerical statistics, e.g., a "liked_by_count" for NSDI (not shown in figure), without the need to carefully read-modify-write the object.

User	Op ID	Operation
Alice	w_1	insert(Alice, "-,Town", NYC)
Bob	r_2	get(Alice, "-,Town")
Bob	w_3	insert(Bob, "-,Town", LA)
Alice	r_4	get(Bob, "-,Town"))
Carol	w_5	insert(Carol, "Likes, NSDI", 8/31/12)
Alice	w_6	insert(Alice, "Likes, NSDI", 9/1/12)
Alice	r_7	get(Carol, "Likes, NSDI")
Alice	w_8	insert(Alice, "Friends, Carol", 9/2/12)

(a)



(b) (c)

Op	Dependencies
w_1	-
w_3	**w_1**
w_5	-
w_6	**w_3** w_1
w_8	**w_6** **w_5** w_3 w_1

(d)

Figure 2: (a) A set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.

2.3 Causal Consistency

A rich data model alone does not provide an intuitive and useful storage system. The storage system's consistency guarantees can restrict the possible ordering and timing of operations throughout the system, helping to simplify the possible behaviors that a programmer must reason about and the anomalies that clients may see.

The strongest forms of consistency (linearizability, serializability, and sequential consistency) are provably incompatible with our low-latency requirement [8, 42], and the weakest (eventual consistency) allows many possible orderings and anomalies. For example, under eventual consistency, after Alice updates her profile, she might not see that update after a refresh. Or, if Alice and Bob are commenting back-and-forth on a blog post, Carol might see a random non-contiguous subset of that conversation.

Fortunately, *causal consistency* can avoid many such inconvenient orderings, including the above examples, while guaranteeing low latency. Interestingly, the motivating example Google used in the presentation of their transactional, linearizable, and non-low-latency system Spanner [22]—where a dissident removes an untrustworthy person from his friends list and then posts politically sensitive speech—only requires causal consistency.

Causal consistency provides a partial order over operations in the system according to the notion of potential causality [2, 38], which is defined by three rules:

- **Thread-of-Execution.** An operation performed by a thread is causally after all of its previous ones.
- **Reads-From.** An operation that reads a value is causally after the operation that wrote the value.
- **Transitive-Closure.** If operation a is causally after b, and b is causally after c, then a is causally after c.

Figure 2 shows several example operations and illustrates their causal relationships. Arrows indicate the sink is causally after the source.

Write operations have *dependencies* on all other write operations that they are causally after. Eiger uses these dependencies to enforce causal consistency: It does not apply (commit) a write in a cluster until verifying that the operation's dependencies are *satisfied*, meaning those writes have already been applied in the cluster.

While the number of dependencies for a write grows with a client's lifetime, the system does not need to track every dependency. Rather, only a small subset of these, the *nearest dependencies*, are necessary for ensuring causal consistency. These dependencies, which have a longest path of one hop to the current operation, transitively capture all of the ordering constraints on this operation. In particular, because all non-nearest dependencies are depended upon by at least one of the nearest, if this current operation occurs after the nearest dependencies, then it will occur after all non-nearest as well (by transitivity). Eiger actually tracks *one-hop dependencies*, a slightly larger superset of nearest dependencies, which have a shortest path of one hop to the current operation. The motivation behind tracking one-hop dependencies is discussed in Section 3.2. Figure 2(d) illustrates the types of dependencies, e.g., w_6's dependency on w_1 is one-hop but not nearest.

3 Eiger System Design

The design of Eiger assumes an underlying partitioned, reliable, and linearizable data store inside of each datacenter. Specifically, we assume:

1. The keyspace is partitioned across logical servers.
2. Linearizability is provided inside a datacenter.
3. Keys are stored on logical servers, implemented with replicated state machines. We assume that a failure does not make a logical server unavailable, unless it makes the entire datacenter unavaible.

Each assumption represents an orthogonal direction of research to Eiger. By assuming these properties instead of specifying their exact design, we focus our explanation on the novel facets of Eiger.

Keyspace partitioning may be accomplished with consistent hashing [34] or directory-based approaches [6, 30]. Linearizability within a datacenter is achieved by partitioning the keyspace and then providing linearizability for each partition [33]. Reliable, linearizable servers can be implemented with Paxos [39] or primary-backup [3] approaches, e.g., chain replication [57]. Many existing systems [5, 13, 16, 54], in fact, provide all assumed properties when used inside a single datacenter.

3.1 Achieving Causal Consistency

Eiger provides causal consistency by explicitly checking that an operation's nearest dependencies have been applied before applying the operation. This approach is similar to the mechanism used by COPS [43], although COPS places dependencies on values, while Eiger uses dependencies on *operations*.

Tracking dependencies on operations significantly improves Eiger's efficiency. In the column family data model, it is not uncommon to simultaneously read or write many columns for a single key. With dependencies on values, a separate dependency must be used for each column's value and thus |column| dependency checks would be required; Eiger could check as few as one. In the worst case, when all columns were written by different operations, the number of required dependency checks degrades to one per value.

Dependencies in Eiger consist of a locator and a unique id. The *locator* is used to ensure that any other operation that depends on this operation knows which node to check with to determine if the operation has been committed. For mutations of individual keys, the locator is simply the key itself. Within a write transaction the locator can be any key in the set; all that matters is that each "sub-operation" within an atomic write be labeled with the same locator.

The *unique id* allows dependencies to precisely map to operations and is identical to the operation's timestamp. A node in Eiger checks dependencies by sending a `dep_check` operation to the node in its local datacenter that owns the locator. The node that owns the locator checks local data structures to see if has applied the operation identified by its unique id. If it has, it responds immediately. If not, it blocks the `dep_check` until it applies the operation. Thus, once all `dep_check`s return, a

server knows all causally previous operations have been applied and it can safely apply this operation.

3.2 Client Library

Clients access their local Eiger datacenter using a client library that: (1) mediates access to nodes in the local datacenter; (2) executes the read and write transaction algorithms; and, most importantly (3) tracks causality and attaches dependencies to write operations.[1]

The client library mediates access to the local datacenter by maintaining a view of its live servers and the partitioning of its keyspace. The library uses this information to send operations to the appropriate servers and sometimes to split operations that span multiple servers.

The client library tracks causality by observing a client's operations.[2] The API exposed by the client library matches that shown earlier in Table 1 with the addition of a `actor_id` field. As an optimization, dependencies are tracked on a per-user basis with the `actor_id` field to avoid unnecessarily adding thread-of-execution dependencies between operations done on behalf of different real-world users (e.g., operations issued on behalf of Alice are not entangled with operations issued on behalf of Bob).

When a client issues a write, the library attaches dependencies on its previous write and on all the writes that wrote a value this client has observed through reads since then. This *one-hop* set of dependencies is the set of operations that have a path of length one to the current operation in the causality graph. The one-hop dependencies are a superset of the nearest dependencies (which have a longest path of length one) and thus attaching and checking them suffices for providing causal consistency.

We elect to track one-hop dependencies because we can do so without storing any dependency information at the servers. Using one-hop dependencies slightly increases both the amount of memory needed at the client nodes and the data sent to servers on writes.[3]

3.3 Basic Operations

Eiger's basic operations closely resemble Cassandra, upon which it is built. The main differences involve the use of server-supplied logical timestamps instead of client-supplied real-time timestamps and, as described above, the use of dependencies and `dep_check`s.

Logical Time. Clients and servers in Eiger maintain a logical clock [38], and messages include a logical timestamp that updates these clocks. The clocks and timestamps provide a progressing logical time throughout the entire system. The low-order bits in each timestamps are set to the stamping server's unique identifier, so each is globally distinct. Servers use these logical timestamps to uniquely identify and order operations.

Local Write Operations. All three write operations in Eiger—`insert`, `add`, and `delete`—operate by replacing the current (potentially non-existent) column in a location. `insert` overwrites the current value with a new column, e.g., update Alice's home town from NYC to MIA. `add` merges the current counter column with the update, e.g., increment a liked-by count from 8 to 9. `delete` overwrites the current column with a tombstone, e.g., Carol is no longer friends with Alice. When each new column is written, it is *timestamped* with the current logical time at the server applying the write.

Cassandra atomically applies updates to a single row using snap trees [14], so all updates to a single key in a `batch_mutate` have the same timestamp. Updates to different rows on the same server in a `batch_mutate` will have different timestamps because they are applied at different logical times.

Read Operations. Read operations return the current column for each requested location. Normal columns return binary data. Deleted columns return an empty column with a deleted bit set. The client library strips deleted columns out of the returned results, but records dependencies on them as required for correctness. Counter columns return a 64-bit integer.

Replication. Servers replicate write operations to their *equivalent* servers in other datacenters. These are the servers that own the same portions of the keyspace as the local server. Because the keyspace partitioning may vary from datacenter to datacenter, the replicating server must sometimes split `batch_mutate` operations.

When a remote server receives a replicated `add` operation, it applies it normally, merging its update with the current value. When a server receives a replicated `insert` or `delete` operation, it compares the timestamps for each included column against the current column for each location. If the replicated column is logically newer, it uses the timestamp from the replicated column and otherwise overwrites the column as it would with a local write. That timestamp, assigned by the

[1] Our implementation of Eiger, like COPS before it, places the client library with the storage system client—typically a web server. Alternative implementations might store the dependencies on a unique node per client, or even push dependency tracking to a rich javascript application running in the client web browser itself, in order to successfully track web accesses through different servers. Such a design is compatible with Eiger, and we view it as worthwhile future work.

[2] Eiger can only track causality it sees, so the traditional criticisms of causality [20] still apply, e.g., we would not capture the causality associated with an out-of-band phone call.

[3] In contrast, our alternative design for tracking the (slightly smaller set of) nearest dependencies put the dependency storage burden on the servers, a trade-off we did not believe generally worthwhile.

datacenter that originally accepted the operation that wrote the value, uniquely identifies the operation. If the replicated column is older, it is discarded. This simple procedure ensures causal consistency: If one column is causally after the other, it will have a later timestamp and thus overwrite the other.

The overwrite procedure also implicitly handles *conflicting operations* that concurrently update a location. It applies the *last-writer-wins rule* [55] to deterministically allow the later of the updates to overwrite the other. This ensures that all datacenters converge to the same value for each column. Eiger could detect conflicts using previous pointers and then resolve them with application-specific functions similar to COPS, but we did not implement such conflict handling and omit details for brevity.

Counter Columns. The commutative nature of counter columns complicates tracking dependencies. In normal columns with overwrite semantics, each value was written by exactly one operation. In counter columns, each value was affected by many operations. Consider a counter with value 7 from +1, +2, and +4 operations. Each operation contributed to the final value, so a read of the counter incurs dependencies on all three. Eiger stores these dependencies with the counter and returns them to the client, so they can be attached to its next write.

Naively, every update of a counter column would increment the number of dependencies contained by that column *ad infinitum*. To bound the number of contained dependencies, Eiger structures the `add` operations occurring within a datacenter. Recall that all locally originating `add` operations within a datacenter are already ordered because the datacenter is linearizable. Eiger explicitly tracks this ordering in a new `add` by adding an *extra dependency* on the previously accepted `add` operation from the datacenter. This creates a single dependency chain that transitively covers all previous updates from the datacenter. As a result, each counter column contains at most one dependency per datacenter.

Eiger further reduces the number of dependencies contained in counter columns to the nearest dependencies *within* that counter column. When a server applies an `add`, it examines the operation's attached dependencies. It first identifies all dependencies that are on updates from other datacenters to this counter column. Then, if any of those dependencies match the currently stored dependency for another datacenter, Eiger drops the stored dependency. The new operation is causally after any local matches, and thus a dependency on it transitively covers those matches as well. For example, if Alice reads a counter with the value 7 and then increments it, her +1 is causally after all operations that commuted to create the 7. Thus, any reads of the resulting 8 would only bring a dependency on Alice's update.

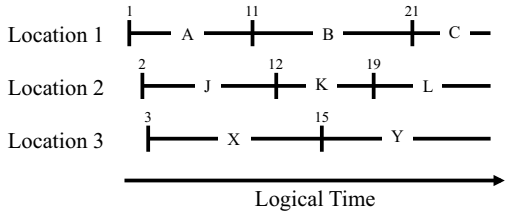


Figure 3: Validity periods for values written to different locations. Crossbars (and the specified numeric times) correspond to the earliest and latest valid time for values, which are represented by letters.

4 Read-Only Transactions

Read-only transactions—the only read operations in Eiger—enable clients to see a consistent view of multiple keys that may be spread across many servers in the local datacenter. Eiger's algorithm guarantees low latency because it takes at most two rounds of parallel non-blocking reads in the local datacenter, plus at most one additional round of local non-blocking checks during concurrent write transactions, detailed in §5.4. We make the same assumptions about reliability in the local datacenter as before, including "logical" servers that do not fail due to linearizable state machine replication.

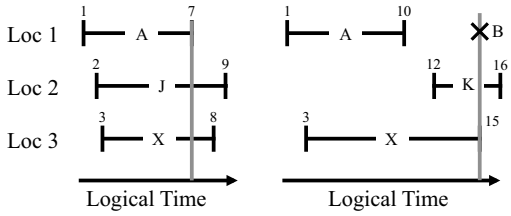
Why read-only transactions? Even though Eiger tracks dependencies to update each datacenter consistently, non-transactional reads can still return an inconsistent set of values. For example, consider a scenario where two items were *written* in a causal order, but read via two separate, parallel reads. The two reads could bridge the write operations (one occurring before either write, the other occurring after both), and thus return values that never actually occurred together, e.g., a "new" object and its "old" access control metadata.

4.1 Read-only Transaction Algorithm

The key insight in the algorithm is that *there exists a consistent result for every query at every logical time*. Figure 3 illustrates this: As operations are applied in a consistent causal order, every data location (key and column) has a consistent value at each logical time.

At a high level, our new read transaction algorithm marks each data location with validity metadata, and uses that metadata to determine if a first round of optimistic reads is consistent. If the first round results are not consistent, the algorithm issues a second round of reads that are guaranteed to return consistent results.

More specifically, each data location is marked with an *earliest valid time* (EVT). The EVT is set to the server's logical time when it locally applies an operation that writes a value. Thus, in an operation's accepting



(a) One Round Sufficient (b) Two Rounds Needed

Figure 4: Examples of read-only transactions. The effective time of each transaction is shown with a gray line; this is the time requested for location 1 in the second round in (b).

datacenter—the one at which the operation originated—the EVT is the same as its timestamp. In other datacenters, the EVT is later than its timestamp. In both cases, the EVT is the exact logical time when the value became visible in the local datacenter.

A server responds to a read with its currently visible value, the corresponding EVT, and its current logical time, which we call the *latest valid time* (LVT). Because this value is still visible, we know it is valid for at least the interval between the EVT and LVT. Once all first-round reads return, the client library compares their times to check for consistency. In particular, it knows all values were valid at the same logical time (i.e., correspond to a consistent snapshot) iff the maximum EVT $\leq$ the minimum LVT. If so, the client library returns these results; otherwise, it proceeds to a second round. Figure 4(a) shows a scenario that completes in one round.

The *effective time* of the transaction is the minimum LVT $\geq$ the maximum EVT. It corresponds both to a logical time in which all retrieved values are consistent, as well as the current logical time (as of its response) at a server. As such, it ensures freshness—necessary in causal consistency so that clients always see a progressing datacenter that reflects their own updates.

For brevity, we only sketch a proof that read transactions return the set of results that were visible in their local datacenter at the transaction's effective time, EffT. By construction, assume a value is visible at logical time t iff val.EVT $\leq t \leq$ val.LVT. For each returned value, if it is returned from the first round, then val.EVT $\leq$ maxEVT $\leq$ EffT by definition of maxEVT and EffT, and val.LVT $\geq$ EffT because it is not being requested in the second round. Thus, val.EVT $\leq$ EffT $\leq$ val.LVT, and by our assumption, the value was visible at EffT. If a result is from the second round, then it was obtained by a second-round read that explicitly returns the visible value at time EffT, described next.

4.2 Two-Round Read Protocol

A read transaction requires a second round if there does not exist a single logical time for which *all* values read in the first round are valid. This can only occur when there are concurrent updates being applied locally to the requested locations. The example in Figure 4(b) requires a second round because location 2 is updated to value K at time 12, which is not before time 10 when location 1's server returns value A.

```
function read_only_trans(requests):
  # Send first round requests in parallel
  for r in requests
    val[r] = multiget_slice(r)
  # Calculate the maximum EVT
  maxEVT = 0
  for r in requests
    maxEVT = max(maxEVT, val[r].EVT)
  # Calculate effective time
  EffT = ∞
  for r in requests
    if val[r].LVT ≥ maxEVT
      EffT = min(EffT, val[r].LVT)
  # Send second round requests in parallel
  for r in requests
    if val[r].LVT < EffT
       val[r] = multiget_slice_by_time(r, EffT)
  # Return only the requested data
  return extract_keys_to_columns(res)
```

Figure 5: Pseudocode for read-only transactions.

During the second round, the client library issues `multiget_slice_by_time` requests, specifying a read at the transaction's effective time. These reads are sent only to those locations for which it does not have a valid result, i.e., their LVT is earlier than the effective time. For example, in Figure 4(b) a `multiget_slice_by_time` request is sent for location 1 at time 15 and returns a new value B.

Servers respond to `multiget_slice_by_time` reads with the value that was valid at the requested logical time. Because that result may be different than the currently visible one, servers sometimes must store old values for each location. Fortunately, the extent of such additional storage can be limited significantly.

4.3 Limiting Old Value Storage

Eiger limits the need to store old values in two ways. First, read transactions have a timeout that specifies their maximum real-time duration. If this timeout fires—which happens only when server queues grow pathologically long due to prolonged overload—the client library restarts a fresh read transaction. Thus, servers only need to store old values that have been overwritten within this timeout's duration.

Second, Eiger retains only old values that could be requested in the second round. Thus, servers store only

values that are *newer* than those returned in a first round within the timeout duration. For this optimization, Eiger stores the last access time of each value.

4.4 Read Transactions for Linearizability

Linearizability (strong consistency) is attractive to programmers when low latency and availability are not strict requirements. Simply being linearizable, however, does not mean that a system is transactional: There may be no way to extract a mutually consistent set of values from the system, much as in our earlier example for read transactions. Linearizability is only defined on, and used with, operations that read or write a single location (originally, shared memory systems) [33].

Interestingly, our algorithm for read-only transactions works for fully linearizable systems, *without* modification. In Eiger, in fact, if all writes that are concurrent with a read-only transaction originated from the local datacenter, the read-only transaction provides a consistent view of that linearizable system (the local datacenter).

5 Write-Only Transactions

Eiger's write-only transactions allow a client to atomically write many columns spread across many keys in the local datacenter. These values also appear atomically in remote datacenters upon replication. As we will see, the algorithm guarantees low latency because it takes at most 2.5 message RTTs in the *local* datacenter to complete, no operations acquire locks, and all phases wait on only the previous round of messages before continuing.

Write-only transactions have many uses. When a user presses a save button, the system can ensure that all of her five profile updates appear simultaneously. Similarly, they help maintain symmetric relationships in social networks: When Alice accepts Bob's friendship request, both friend associations appear at the same time.

5.1 Write-Only Transaction Algorithm

To execute an `atomic_mutate` request—which has identical arguments to `batch_mutate`—the client library splits the operation into one sub-request per local server across which the transaction is spread. The library randomly chooses one key in the transaction as the *coordinator key*. It then transmits each sub-request to its corresponding server, annotated with the coordinator key.

Our write transaction is a variant of two-phase commit [51], which we call *two-phase commit with positive cohorts and indirection* (2PC-PCI). 2PC-PCI operates differently depending on whether it is executing in the original (or "accepting") datacenter, or being applied in the remote datacenter after replication.

There are three differences between traditional 2PC and 2PC-PCI, as shown in Figure 6. First, 2PC-PCI has only positive cohorts; the coordinator always commits the transaction once it receives a vote from all cohorts.[4] Second, 2PC-PCI has a different pre-vote phase that varies depending on the origin of the write transaction. In the accepting datacenter (we discuss the remote below), the client library sends each participant its sub-request directly, and this transmission serves as an implicit PREPARE message for each cohort. Third, 2PC-PCI cohorts that cannot answer a query—because they have voted but have not yet received the commit—ask the coordinator if the transaction is committed, effectively *indirecting* the request through the coordinator.

5.2 Local Write-Only Transactions

When a *participant* server, which is either the coordinator or a cohort, receives its transaction sub-request from the client, it prepares for the transaction by writing each included location with a special "pending" value (retaining old versions for second-round reads). It then sends a YESVOTE to the coordinator.

When the coordinator receives a YESVOTE, it updates its count of prepared keys. Once all keys are prepared, the coordinator commits the transaction. The coordinator's current logical time serves as the (global) timestamp and (local) EVT of the transaction and is included in the COMMIT message.

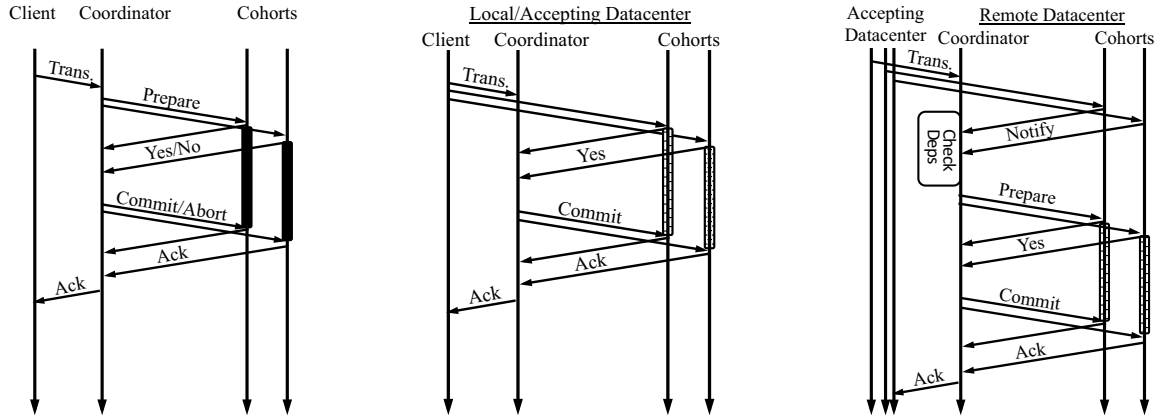
When a cohort receives a COMMIT, it replaces the "pending" columns with the update's real values, and ACKs the committed keys. Upon receiving all ACKs, the coordinator safely cleans up its transaction state.

5.3 Replicated Write-Only Transactions

Each transaction sub-request is replicated to its "equivalent" participant(s) in the remote datacenter, possibly splitting the sub-requests to match the remote key partitioning. When a cohort in a remote datacenter receives a sub-request, it sends a NOTIFY with the key count to the transaction coordinator in its datacenter. This coordinator issues any necessary `dep_check`s upon receiving its own sub-request (which contains the coordinator key). The coordinator's checks cover the entire transaction, so cohorts send no checks. Once the coordinator has received all NOTIFY messages and `dep_check`s responses, it sends each cohort a PREPARE, and then proceeds normally.

For reads received during the *indirection window* in which participants are uncertain about the status of a

[4] Eiger only has positive cohorts because it avoids all the normal reasons to abort (vote no): It does not have general transactions that can force each other to abort, it does not have users that can cancel operations, and it assumes that its logical servers do not fail.



(a) Traditional 2PC (b) Local Write-Only Txn (c) Replicated Write-Only Txn

Figure 6: Message flow diagrams for traditional 2PC and write-only transaction. Solid boxes denote when cohorts block reads. Striped boxes denote when cohorts will indirect a commitment check to the coordinator.

transaction, cohorts must query the coordinator for its state. To minimize the duration of this window, before preparing, the coordinator waits for (1) *all* participants to NOTIFY and (2) all `dep_check`s to return. This helps prevent a slow replica from causing needless indirection.

Finally, replicated write-only transactions differ in that participants do not always write pending columns. If a location's current value has a newer timestamp than that of the transaction, the validity interval for the transaction's value is empty. Thus, no read will ever return it, and it can be safely discarded. The participant continues in the transaction for simplicity, but does not need to indirect reads for this location.

5.4 Reads when Transactions are Pending

If a first-round read accesses a location that could be modified by a pending transaction, the server sends a special empty response that only includes a LVT (i.e., its current time). This alerts the client that it must choose an effective time for the transaction and send the server a second-round `multiget_slice_by_time` request.

When a server with pending transactions receives a `multiget_slice_by_time` request, it first traverses its old versions for each included column. If there exists a version valid at the requested time, the server returns it.

Otherwise, there are pending transactions whose *potential commit window* intersects the requested time and the server must resolve their ordering. It does so by sending a `commit_check` with this requested time to the transactions' coordinator(s). Each coordinator responds whether the transaction had been committed at that (past) time and, if so, its commit time.

Once a server has collected all `commit_check` responses, it updates the validity intervals of all versions of all relevant locations, up to at least the requested (effective) time. Then, it can respond to the `multiget_slice_by_time` message as normal.

The complementary nature of Eiger's transactional algorithms enables the atomicity of its writes. In particular, the single commit time for a write transaction (EVT) and the single effective time for a read transaction lead each to appear at a single logical time, while its two-phase commit ensures all-or-nothing semantics.

6 Failure

In this section, we examine how Eiger behaves under failures, including single server failure, meta-client redirection, and entire datacenter failure.

Single server failures are common and unavoidable in practice. Eiger guards against their failure with the construction of logical servers from multiple physical servers. For instance, a logical server implemented with a three-server Paxos group can withstand the failure of one of its constituent servers. Like any system built on underlying components, Eiger inherits the failure modes of its underlying building blocks. In particular, if a logical server assumes no more than f physical machines fail, Eiger must assume that within a single logical server no more than f physical machines fail.

Meta-clients that are the clients of Eiger's clients (i.e., web browsers that have connections to front-end web tier machines) will sometimes be directed to a different datacenter. For instance, a redirection may occur when there is a change in the DNS resolution policy of a service. When a redirection occurs during the middle of an active connection, we expect service providers to detect it using cookies and then redirect clients to their original datacenter (e.g., using HTTP redirects or triangle routing).

When a client is not actively using the service, however, policy changes that reassign it to a new datacenter can proceed without complication.

Datacenter failure can either be transient (e.g., network or power cables are cut) or permanent (e.g., datacenter is physically destroyed by an earthquake). Permanent failures will result in data loss for data that was accepted and acknowledged but not yet replicated to any other datacenter. The colocation of clients inside the datacenter, however, will reduce the amount of externally visible data loss. Only data that is not yet replicated to another datacenter, but has been acknowledged to both Eiger's clients and meta-clients (e.g., when the browser receives an Ajax response indicating a status update was posted) will be visibly lost. Transient datacenter failure will not result in data loss.

Both transient and permanent datacenter failures will cause meta-clients to reconnect to different datacenters. After some configured timeout, we expect service providers to stop trying to redirect those meta-clients to their original datacenters and to connect them to a new datacenter with an empty context. This could result in those meta-clients effectively moving backwards in time. It would also result in the loss of causal links between the data they observed in their original datacenter and their new writes issued to their new datacenter. We expect that transient datacenter failure will be rare (no ill effects), transient failure that lasts long enough for redirection to be abandoned even rarer (causality loss), and permanent failure even rarer still (data loss).

7 Evaluation

This evaluation explores the overhead of Eiger's stronger semantics compared to eventually-consistent Cassandra, analytically compares the performance of COPS and Eiger, and shows that Eiger scales to large clusters.

7.1 Implementation

Our Eiger prototype implements everything described in the paper as 5000 lines of Java added to and modifying the existing 75000 LoC in Cassandra 1.1 [17, 37]. All of Eiger's reads are transactional. We use Cassandra configured for wide-area eventual consistency as a baseline for comparison. In each local cluster, both Eiger and Cassandra use consistent hashing to map each key to a single server, and thus trivially provide linearizability.

In unmodified Cassandra, for a single logical request, the client sends all of its sub-requests to a single server. This server splits `batch_mutate` and `multiget_slice` operations from the client that span multiple servers, sends them to the appropriate server, and re-assembles the responses for the client. In Eiger, the client library handles this splitting, routing, and re-assembly directly, allowing Eiger to save a local RTT in latency and potentially many messages between servers. With this change, Eiger outperforms unmodified Cassandra in most settings. Therefore, to make our comparison to Cassandra fair, we implemented an analogous client library that handles the splitting, routing, and re-assembly for Cassandra. The results below use this optimization.

7.2 Eiger Overheads

We first examine the overhead of Eiger's causal consistency, read-only transactions, and write-only transactions. This section explains why each potential source of overhead does not significantly impair throughput, latency, or storage; the next sections confirm empirically.

Causal Consistency Overheads. Write operations carry *dependency metadata*. Its impact on throughput and latency is low because each dependency is 16B; the number of dependencies attached to a write is limited to its small set of one-hop dependencies; and writes are typically less frequent. Dependencies have no storage cost because they are not stored at the server.

Dependency check operations are issued in remote datacenters upon receiving a replicated write. Limiting these checks to the write's one-hop dependencies minimizes throughput degradation. They do not affect client-perceived latency, occuring only during asynchronous replication, nor do they add storage overhead.

Read-only Transaction Overheads. *Validity-interval metadata* is stored on servers and returned to clients with read operations. Its effect is similarly small: Only the 8B EVT is stored, and the 16B of metadata returned to the client is tiny compared to typical key/column/value sets.

If *second-round reads* were always needed, they would roughly double latency and halve throughput. Fortunately, they occur only when there are concurrent writes to the requested columns in the local datacenter, which is rare given the short duration of reads and writes.

Extra-version storage is needed at servers to handle second-round reads. It has no impact on throughput or latency, and its storage footprint is small because we aggressively limit the number of old versions (see §4.3).

Write-only Transaction Overheads. Write transactions *write columns twice*: once to mark them pending and once to write the true value. This accounts for about half of the moderate overhead of write transactions, evaluated in §7.5. When only some writes are transactional and when the writes are a minority of system operations (as found in prior studies [7, 28]), this overhead has a

	Latency (ms)			
	50%	90%	95%	99%
Reads				
Cassandra-Eventual	0.38	0.56	0.61	1.13
Eiger 1 Round	0.47	0.67	0.70	1.27
Eiger 2 Round	0.68	0.94	1.04	1.85
Eiger Indirected	0.78	1.11	1.18	2.28
Cassandra-Strong-A	85.21	85.72	85.96	86.77
Cassandra-Strong-B	21.89	22.28	22.39	22.92
Writes				
Cassandra-Eventual Cassandra-Strong-A	0.42	0.63	0.91	1.67
Eiger Normal	0.45	0.67	0.75	1.92
Eiger Normal (2)	0.51	0.79	1.38	4.05
Eiger Transaction (2)	0.73	2.28	2.94	4.39
Cassandra-Strong-B	21.65	21.85	21.93	22.29

Table 2: Latency micro-benchmarks.

small effect on overall throughput. The second write overwrites the first, consuming no space.

Many *2PC-PCI messages* are needed for the write-only algorithm. These messages add 1.5 local RTTs to latency, but have little effect on throughput: the messages are small and can be handled in parallel with other steps in different write transactions.

Indirected second-round reads add an extra local RTT to latency and reduce read throughput vs. normal second-round reads. They affect throughput minimally, however, because they occur rarely: only when the second-round read arrives when there is a not-yet-committed write-only transaction on an overlapping set of columns that prepared before the read-only transaction's effective time.

7.3 Experimental Setup

The first experiments use the shared VICCI testbed [45, 58], which provides users with Linux VServer instances. Each physical machine has 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, and 2x1GigE network ports.

All experiments are between multiple VICCI sites. The latency micro-benchmark uses a minimal wide-area setup with a cluster of 2 machines at the Princeton, Stanford, and University of Washington (UW) VICCI sites. All other experiments use 8-machine clusters in Stanford and UW and an additional 8 machines in Stanford as clients. These clients fully load their local cluster, which replicates its data to the other cluster.

The inter-site latencies were 88ms between Princeton and Stanford, 84ms between Princeton and UW, and 20ms between Stanford and UW. Inter-site bandwidth was not a limiting factor.

Every datapoint in the evaluation represents the median of 5+ trials. Latency micro-benchmark trials are 30s, while all other trials are 60s. We elide the first and last quarter of each trial to avoid experimental artifacts.

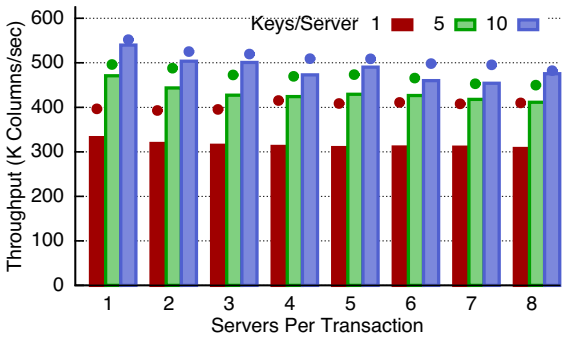


Figure 7: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.

7.4 Latency Micro-benchmark

Eiger always satisfies client operations within a local datacenter and thus, fundamentally, is low-latency. To demonstrate this, verify our implementation, and compare with strongly-consistent systems, we ran an experiment to compare the latency of read and write operations in Eiger vs. three Cassandra configurations: eventual (R=1, W=1), strong-A (R=3, W=1), and strong-B (R=2, W=2), where R and W indicate the number of datacenters involved in reads and writes.[5]

The experiments were run from UW with a single client thread to isolate latency differences. Table 2 reports the median, 90%, 95%, and 99% latencies from operations on a single 1B column. For comparison, two 1B columns, stored on different servers, were also updated together as part of transactional and non-transactional "Eiger (2)" write operations.

All reads in Eiger—one-round, two-round, and worst-case two-round-and-indirected reads—have median latencies under 1ms and 99% latencies under 2.5ms. `atomic_mutate` operations are slightly slower than `batch_mutate` operations, but still have median latency under 1ms and 99% under 5ms. Cassandra's strongly consistent operations fared much worse. Configuration "A" achieved fast writes, but reads had to access all datacenters (including the ~84ms RTT between UW and Princeton); "B" suffered wide-area latency for both reads and writes (as the second datacenter needed for a quorum involved a ~20ms RTT between UW and Stanford).

7.5 Write Transaction Cost

Figure 7 shows the throughput of write-only transactions, and Cassandra's non-atomic batch mutates, when the

[5]Cassandra single-key writes are not atomic across different nodes, so its strong consistency requires read repair (write-back) and $R{>}N/2$.

Parameter	Range	Default	Facebook		
			50%	90%	99%
Value Size (B)	1-4K	128	16	32	4K
Cols/Key for Reads	1-32	5	1	2	128
Cols/Key for Writes	1-32	5	1	2	128
Keys/Read	1-32	5	1	16	128
Keys/Write	1-32	5		1	
Write Fraction	0-1.0	.1		.002	
Write Txn Fraction	0-1.0	.5		0 or 1.0	
Read Txn Fraction	1.0	1.0		1.0	

Table 3: Dynamic workload generator parameters. Range is the space covered in the experiments; Facebook describes the distribution for that workload.

keys they touch are spread across 1 to 8 servers. The experiment used the default parameter settings from Table 3 with 100% writes and 100% write transactions.

Eiger's throughput remains competitive with batch mutates as the transaction is spread across more servers. Additional servers only increase 2PC-PCI costs, which account for less than 10% of Eiger's overhead. About half of the overhead of write-only transactions comes from double-writing columns; most of the remainder is due to extra metadata. Both absolute and Cassandra-relative throughput increase with the number of keys written per server, as the coordination overhead remains independent of the number of columns.

7.6 Dynamic Workloads

We created a dynamic workload generator to explore the space of possible workloads. Table 3 shows the range and default value of the generator's parameters. The results from varying each parameter while the others remain at their defaults are shown in Figure 8.

Space constraints permit only a brief review of these results. Overhead decreases with increasing value size, because metadata represents a smaller portion of message size. Overhead is relatively constant with increases in the columns/read, columns/write, keys/read, and keys/write ratios because while the amount of metadata increases, it remains in proportion to message size. Higher fractions of write transactions (within an overall 10% write workload) do not increase overhead.

Eiger's throughput is overall competitive with the eventually-consistent Cassandra baseline. With the default parameters, its overhead is 15%. When they are varied, its overhead ranges from 0.5% to 25%.

7.7 Facebook Workload

For one realistic view of Eiger's overhead, we parameterized a synthetic workload based upon Facebook's production TAO system [53]. Parameters for value sizes,

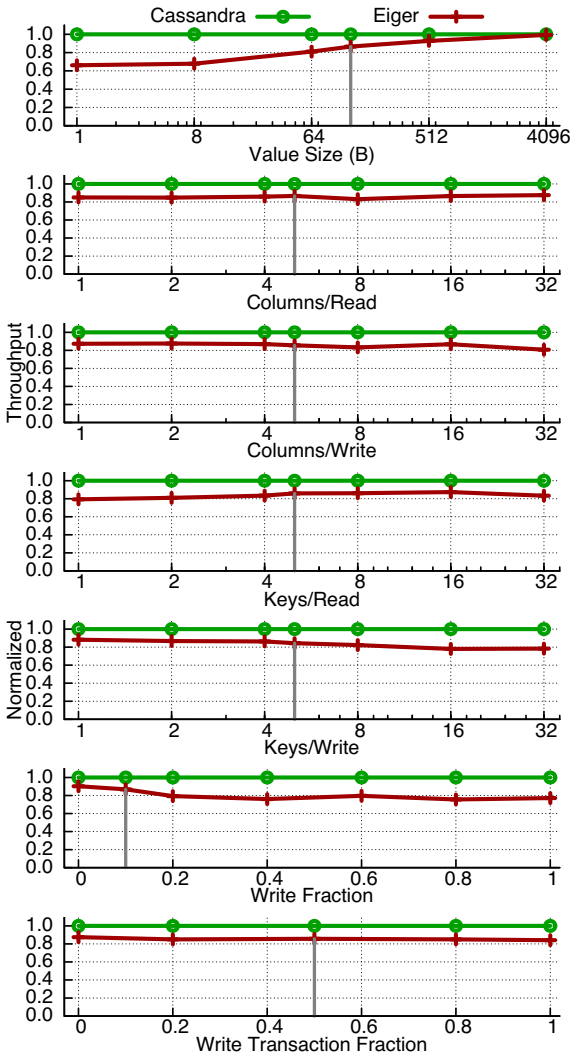


Figure 8: Results from exploring our dynamic-workload generator's parameter space. Each experiment varies one parameter while keeping all others at their default value (indicated by the vertical line). Eiger's throughput is normalized against eventually-consistent Cassandra.

columns/key, and keys/operation are chosen from discrete distributions measured by the TAO team. We show results with a 0% write transaction fraction (the actual workload, because TAO lacks transactions), and with 100% write transactions. Table 3 shows the heavy-tailed distributions' 50th, 90th, and 99th percentiles.

Table 4 shows that the throughput for Eiger is within 7% of eventually-consistent Cassandra. The results for 0% and 100% write transactions are effectively identical because writes are such a small part of the workload. For this real-world workload, Eiger's causal consistency and stronger semantics do not impose significant overhead.

	Ops/sec	Keys/sec	Columns/sec
Cassandra	23,657	94,502	498,239
Eiger	22,088	88,238	466,844
Eiger All Txns	22,891	91,439	480,904
Max Overhead	6.6%	6.6%	6.3%

Table 4: Throughput for the Facebook workload.

7.8 Performance vs. COPS

COPS and Eiger provide different data models and are implemented in different languages, so a direct empirical comparison is not meaningful. We can, however, intuit how Eiger's algorithms perform in the COPS setting.

Both COPS and Eiger achieve low latency around 1ms. Second-round reads would occur in COPS and Eiger equally often, because both are triggered by the same scenario: concurrent writes in the local datacenter to the same keys. Eiger experiences some additional latency when second-round reads are indirected, but this is rare (and the total latency remains low). Write-only transactions in Eiger would have higher latency than their non-atomic counterparts in COPS, but we have also shown their latency to be very low.

Beyond having write transactions, which COPS did not, the most significant difference between Eiger and COPS is the efficiency of read transactions. COPS's read transactions ("COPS-GT") add significant dependency-tracking overhead vs. the COPS baseline under certain conditions. In contrast, by tracking only one-hop dependencies, Eiger avoids the metadata explosion that COPS' read-only transactions can suffer. We expect that Eiger's read transactions would operate roughly as quickly as COPS' non-transactional reads, and the system as a whole would outperform COPS-GT despite offering both read- and write-only transactions and supporting a much more rich data model.

7.9 Scaling

To demonstrate the scalability of Eiger we ran the Facebook TAO workload on N client machines that are fully loading an N-server cluster that is replicating writes to another N-server cluster, i.e., the N=128 experiment involves 384 machines. This experiment was run on PRObE's Kodiak testbed [47], which provides an Emulab [59] with exclusive access to hundreds of machines. Each machine has 2 AMD Opteron 252 CPUS, 8GM RAM, and an InfiniBand high-speed interface. The bottleneck in this experiment is server CPU.

Figure 9 shows the throughput for Eiger as we scale N from 1 to 128 servers/cluster. The bars show throughput normalized against the throughput of the 1-server cluster. Eiger scales out as the number of servers increases, though this scaling is not linear from 1 to 8 servers/cluster.

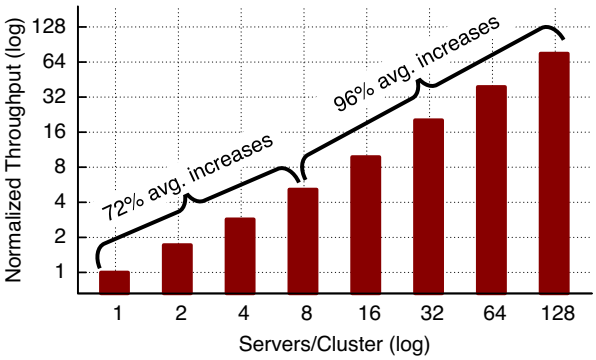


Figure 9: Normalized throughput of N-server clusters for the Facebook TAO workload. Bars are normalized against the 1-server cluster.

The 1-server cluster benefits from batching; all operations that involve multiple keys are executed on a single machine. Larger clusters distribute these multi-key operations over multiple servers and thus lose batching. This mainly affects scaling from 1 to 8 servers/cluster (72% average increase) and we see almost perfect linear scaling from 8 to 128 servers/cluster (96% average increase).

8 Related Work

A large body of research exists about stronger consistency in the wide area. This includes classical research about two-phase commit protocols [51] and distributed consensus (e.g., Paxos [39]). As noted earlier, protocols and systems that provide the strongest forms of consistency are provably incompatible with low latency [8, 42]. Recent examples includes Megastore [9], Spanner [22], and Scatter [31], which use Paxos in the wide-area; PNUTS [21], which provides sequential consistency on a per-key basis and must execute in a key's specified primary datacenter; and Gemini [40], which provides RedBlue consistency with low latency for its blue operations, but high latency for its globally-serialized red operations. In contrast, Eiger guarantees low latency.

Many previous system designs have recognized the utility of causal consistency, including Bayou [44], lazy replication [36], ISIS [12], causal memory [2], and PRACTI [10]. All of these systems require single-machine replicas (datacenters) and thus are not scalable.

Our previous work, COPS [43], bears the closest similarity to Eiger, as it also uses dependencies to provide causal consistency, and targets low-latency and scalable settings. As we show by comparing these systems in Table 5, however, Eiger represents a large step forward from COPS. In particular, Eiger supports a richer data model, has more powerful transaction support (whose algorithms also work with other consistency models), transmits and stores fewer dependencies, eliminates the need

	COPS	COPS-GT	Eiger
Data Model	Key Value	Key Value	**Column Fam**
Consistency	**Causal**	**Causal**	**Causal**
Read-Only Txn	No	**Yes**	**Yes**
Write-Only Txn	No	No	**Yes**
Txn Algos Use	-	Deps	**Logic. Time**
Deps On	Values	Values	**Operations**
Transmitted Deps	**One-Hop**	All-GarbageC	**One-Hop**
Checked Deps	One-Hop	**Nearest**	One-Hop
Stored Deps	**None**	All-GarbageC	**None**
GarbageC Deps	**Unneeded**	Yes	**Unneeded**
Versions Stored	**One**	Few	**Fewer**

Table 5: Comparing COPS and Eiger.

for garbage collection, stores fewer old versions, and is not susceptible to availability problems from metadata explosion when datacenters either fail, are partitioned, or suffer meaningful slow-down for long periods of time.

The database community has long supported consistency across multiple keys through general transactions. In many commercial database systems, a single primary executes transactions across keys, then lazily sends its transaction log to other replicas, potentially over the wide-area. In scale-out designs involving data partitioning (or "sharding"), these transactions are typically limited to keys residing on the same server. Eiger does not have this restriction. More fundamentally, the single primary approach inhibits low-latency, as write operations must be executed in the primary's datacenter.

Several recent systems reduce the inter-datacenter communication needed to provide general transactions. These include Calvin [56], Granola [24], MDCC [35], Orleans [15], and Walter [52]. In their pursuit of general transactions, however, these systems all choose consistency models that cannot guarantee low-latency operations. MDCC and Orleans acknowledge this with options to receive fast-but-potentially-incorrect responses.

The implementers of Sinfonia [1], TxCache [46], HBase [32], and Spanner [22], also recognized the importance of limited transactions. Sinfonia provides "mini" transactions to distributed shared memory and TXCache provides a consistent but potentially stale cache for a relational database, but both only considers operations within a single datacenter. HBase includes read- and write-only transactions within a single "region," which is a subset of the capacity of a single node. Spanner's read-only transactions are similar to the original distributed read-only transactions [18], in that they always take at least two rounds and block until all involved servers can guarantee they have applied all transactions that committed before the read-only transaction started. In comparison, Eiger is designed for geo-replicated storage, and its transactions can execute across large cluster of nodes, normally only take one round, and never block.

The widely used MVCC algorithm [11, 49] and Eiger maintain multiple versions of objects so they can provide clients with a consistent view of a system. MVCC provides full snapshot isolation, sometimes rejects writes, has state linear in the number of recent reads and writes, and has a sweeping process that removes old versions. Eiger, in contrast, provides only read-only transactions, never rejects writes, has at worst state linear in the number of recent writes, and avoids storing most old versions while using fast timeouts for cleaning the rest.

9 Conclusion

Impossibility results divide geo-replicated storage systems into those that can provide the strongest forms of consistency and those that can guarantee low latency. Eiger represents a new step forward on the low latency side of that divide by providing a richer data model and stronger semantics. Our experimental results demonstrate that the overhead of these properties compared to a non-transactional eventually-consistent baseline is low, and we expect that further engineering and innovations will reduce it almost entirely.

This leaves applications with two choices for geo-replicated storage. Strongly-consistent storage is required for applications with global invariants, e.g., banking, where accounts cannot drop below zero. And Eiger-like systems can serve all other applications, e.g., social networking (Facebook), encyclopedias (Wikipedia), and collaborative filtering (Reddit). These applications no longer need to settle for eventual consistency and can instead make sense of their data with causal consistency, read-only transactions, and write-only transactions.

Acknowledgments. The authors would like to thank the NSDI program committee and especially our shepherd, Ethan Katz-Bassett, for their helpful comments. Sid Sen, Ariel Rabkin, David Shue, and Xiaozhou Li provided useful comments on this work; Sanjeev Kumar, Harry Li, Kaushik Veeraraghavan, Jack Ferris, and Nathan Bronson helped us obtain the workload characteristics of Facebook's TAO system; Sapan Bhatia and Andy Bavier helped us run experiments on the VICCI testbed; and Gary Sandine and Andree Jacobson helped with the PRObE Kodiak testbed.

This work was supported by funding from National Science Foundation Awards CSR-0953197 (CAREER), CCF-0964474, MRI-1040123 (VICCI), CNS-1042537 and 1042543 (PRObE), and the Intel Science and Technology Center for Cloud Computing.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 27(3), 2009.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9 (1), 1995.

[3] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Conf. Software Engineering*, Oct. 1976.

[4] Amazon. Simple storage service. http://aws.amazon.com/s3/, 2012.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, Oct. 2009.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[8] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[9] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Jan. 2011.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, May 2006.

[11] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computer Surveys*, 13(2), June 1981.

[12] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[13] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[14] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, Jan. 2010.

[15] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SOCC*, 2011.

[16] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[17] Cassandra. http://cassandra.apache.org/, 2012.

[18] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Trans. Info. Theory*, 11(2), 1985.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2), 2008.

[20] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, Dec. 1993.

[21] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, Oct 2012.

[23] CouchDB. http://couchdb.apache.org/, 2012.

[24] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, Jun 2012.

[25] P. Dixon. Shopzilla site redesign: We get what we measure. Velocity Conference Talk, 2009.

[26] eBay. Personal communication, 2012.

[27] Facebook. Personal communication, 2011.

[28] J. Ferris. The TAO graph database. CMU PDL Talk, April 2012.

[29] B. Fitzpatrick. Memcached: a distributed memory object caching system. http://memcached.org/, 2011.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[31] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, Oct. 2011.

[32] HBase. http://hbase.apache.org/, 2012.

[33] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[34] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[35] T. Kraska, G. Pang, M. J. Franklin, and S. Madden. MDCC: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.

[36] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[37] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. In *LADIS*, Oct. 2009.

[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[39] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[40] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, Oct 2012.

[41] G. Linden. Make data useful. Stanford CS345 Talk, 2006.

[42] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, Oct. 2011.

[44] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[45] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci., 2011.

[46] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, Oct. 2010.

[47] PRObE. http://www.nmc-probe.org/, 2013.

[48] Redis. http://redis.io/, 2012.

[49] D. P. Reed. *Naming and Synchronization in a Decentralized Computer Systems*. PhD thesis, Mass. Inst. of Tech., 1978.

[50] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. Velocity Conference Talk, 2009.

[51] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Info. Theory*, 9(3), May 1983.

[52] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, Oct. 2011.

[53] TAO. A read-optimized globally distributed store for social graph data. Under Submission, 2012.

[54] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX ATC*, June 2009.

[55] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Sys.*, 4(2), 1979.

[56] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, May 2012.

[57] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, Dec. 2004.

[58] VICCI. http://vicci.org/, 2012.

[59] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.