K2: Reading Quickly from Storage Across Many Datacenters

Khiem Ngo, Haonan Lu, Wyatt Lloyd
Princeton University

***Abstract*—The infrastructure available to large-scale and medium-scale web services now spans dozens of geographically dispersed datacenters. Deploying across many datacenters has the potential to significantly reduce end-user latency by serving users nearer their location. However, deploying across many datacenters requires the backend storage system be partially replicated. In turn, this can sacrifice the low latency benefits of many datacenters, especially when a storage system provides guarantees on what operations will observe.**

We present the K2 storage system that provides lower latency for large-scale and medium-scale web services using partial replication of data over many datacenters with strong guarantees: causal consistency, read-only transactions, and write-only transactions. K2 provides the best possible worst-case latency for partial replication, a single round trip to remote datacenters, and often avoids sending *any* requests to far away datacenters using a novel replication approach, write-only transaction algorithm, and read-only transaction algorithm.

I. Introduction

The infrastructure necessary for large-scale web services and available to medium-scale web services now includes resources in many geographically dispersed datacenters. The sheer size of large-scale services requires they deploy across many datacenters—Google has 24 datacenters [27] and Facebook has 15 datacenters [21]. Medium-scale services that can be fully deployed in a few datacenters also have the option of deploying across many datacenters due to the proliferation of available locations on cloud platforms [6], [26], [44].

Deploying a service across many datacenters has the potential to significantly decrease its latency for end-users through increased proximity. For instance, users of a social network in Australia can have significantly faster interactions with the service when their requests are handled entirely in Australia instead of needing to go to another continent. This requires that both the frontend web server that is handling the user's requests be in Australia and the backend storage system that holds the data of the social network—e.g., friend lists, status updates—be in Australia. Spreading a service's frontend web servers across many datacenters does not change the required number of web servers because each does disjoint work; 100 servers in 1 datacenter and 10 servers in each of 10 datacenters can handle the same number of user requests. Fully replicating a backend storage system across many datacenters, however, proportionally increases its costs because each replica in each datacenter does the same work of storing and serving all the data. This makes full replication across many datacenters economically infeasible. Instead, the large-scale services that must use many datacenters *partially replicate* their data by storing only a subset of it in each datacenter [7], [20].

Partial replication of data, however, can eliminate the latency benefits of many datacenters and even increase a service's latency compared to full replication over a few datacenters. The latency benefit is eliminated if a datacenter does not have the required data for a request and needs to go to a far-away datacenter once. A service's latency is *worse* if the storage system needs to go to a far-away datacenter more than once: it would have been faster to send the user's request to that far-away datacenter and handle all its backend accesses there. We contend this is why medium-scale web services typically stick to full replication over a few datacenters.

Further complicating matters are the guarantees the data store provides. These guarantees include *consistency*, i.e., what interleavings of write operations are visible to reads; and *transactions*, i.e., what operations can appear to be grouped into an atomic block. These guarantees enable and simplify correct application logic, for example, by ensuring a referent and reference appear in the correct order, as well as reduce user-visible anomalies. In storage systems that provide such guarantees over partially-replicated data, multiple round trips to far-away datacenters would be common, leading to even higher latency experienced by end-users and dwarfing the benefits of having many datacenters closer to them (§II).

We present K2, a storage system that provides lower latency for large-scale and medium-scale web services using partial replication of data over many datacenters. K2 provides guarantees that achieve a sweet spot in the tradeoff between the strength of abstraction and low latency: causal consistency, read-only transactions, and write-only transactions. Prior work that supports stronger guarantees is incompatible with low latency; while prior work that achieves low latency does not support any type of transactions.

K2 unlocks low latency for these guarantees by realizing two design goals. First, it has at most one parallel round of non-blocking requests to far-away datacenters. Second, it often avoids sending *any* requests to far away datacenters. The first goal ensures K2 has latency no worse than fully replicating across a few datacenters while the second goal provides lower latency for most requests.

K2's design includes several components that work together to achieve these goals. First, K2 fully replicates metadata and runs its algorithms primarily on that metadata. This enables it to directly use an existing mechanism for causal consistency: one-hop dependency checking [39]. Providing transactions,

 DOI 10.1109/DSN48987.2021.00034

however, remains challenging because existing mechanisms do not achieve either of our design goals. Existing write-only transactions algorithms cause cross-datacenter requests to block and result in latency worse than a fully-replicated system. To bound its worst case, K2 introduces a constrained replication topology and a new write-only transaction algorithm. Constraining replication ensures each datacenter knows where to read consistent values. The new write-only transaction algorithm decouples the visibility of data for local reads from remote reads to ensure local reads remain consistent while ensuring remote reads never block.

To provide local datacenter latency in the common case, K2 integrates a small cache in each datacenter and introduces a cache-aware read-only transaction algorithm. The algorithm maximizes its ability to use cached data while ensuring consistency and isolation. This enables read-only transactions to often be handled locally with zero cross-datacenter requests. K2 also uses the cache to provide low latency for write-only transactions by committing them locally.

Our evaluation of K2 compares to an adaption of a fully replicated system to work with partial replication and a concurrently developed system that provides causal consistency with transactions over partially replicated data. We find that K2 has significantly lower latency than the baselines in all evaluated settings.

In summary, the primary contribution of this paper is the first design that realizes the low latency benefit of many datacenters for the strong guarantees of causal consistency, read-only, and write-only transactions. Read-only transactions achieve low latency because they require zero cross-datacenter requests in the common case and one round of non-blocking requests in the worst case. K2's design achieves this through its novel replication approach, write-only transaction algorithm, and read-only transaction algorithm.

II. Background and Motivation

This section provides background, motivates partial replication, and identifies our design goals.

A. Background

Figure 1 shows the general structure of web services. They are distributed across several datacenters with frontends that handle user requests by executing application code that reads and writes data from a backend storage system. To match common terminology, we use *clients* to refer to frontends and *servers* to refer to backend storage servers.

Together, the backend storage servers provide a set of programming abstractions and accompanying guarantees to the frontend application servers for manipulating an application's data. Stronger guarantees, such as transactions, simplify application development by reducing the number of states and edge cases that a programmer needs to reason about.

Targeted Guarantees. We target guarantees that provide a sweet spot in the tradeoff between the strength of the abstraction provided by a storage system and its latency: causal consistency, read-only transactions, and write-only transactions.

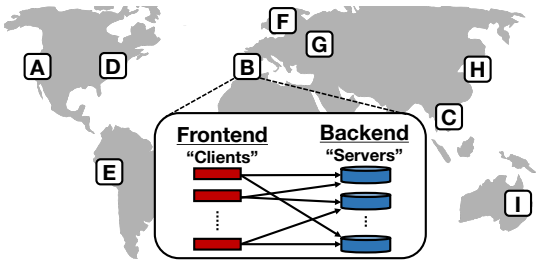


Fig. 1: Web services are made up of frontends and backend storage servers spread across datacenters. Large services deploy across many datacenters, e.g., A–I. Medium services often only deploy across a few datacenters, e.g., A–C. K2 improves latency for large services and enables medium services to also improve their latency by using many datacenters.

Stronger guarantees require cross-datacenter requests [10], [15], [25], [37] and thus cannot reap the low latency benefits of many datacenters. Weaker guarantees are harder to program against without being necessary for low latency.

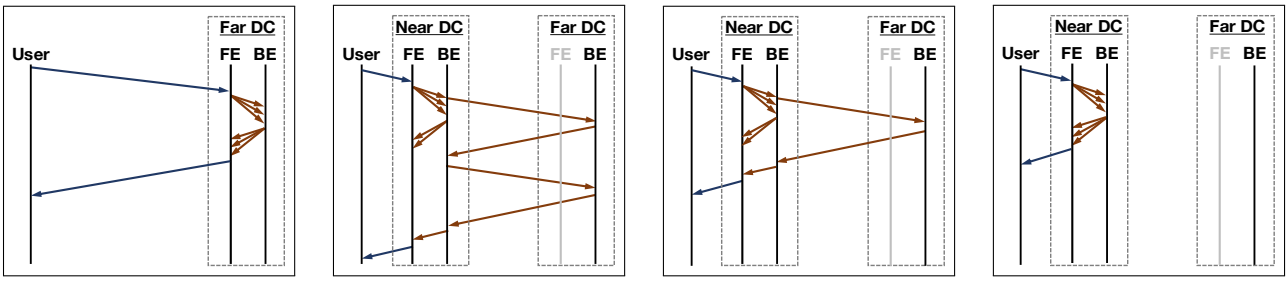
This makes K2 suitable for the large set of applications that prioritize low latency over the strongest guarantees (e.g., read-write transactions with strict serializability) such as social networks, collaborative filtering, and encyclopedias. It is even suitable for sensitive applications such as access-control: K2's guarantees are sufficiently strong for Zanzibar, Google's global authorization system [45].

Causal consistency provides a partial order over operations that ensures causality, which has three rules [2], [35]: if an operation a happens before an operation b in the same thread of execution, then $a \rightarrow b$, meaning that b is causally after a; if an operation a writes the value v to the variable x, and an operation b reads the value v from x, then $a \rightarrow b$; if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Read-only transactions are a group of read operations that appear to all read from a single consistent state of the data store. The read operations can span data that is spread across many different shards of the data store. Grouping them in a transaction in effect combines them into a single operation. In K2, the group of reads will see the same, causally-consistent state of the data store.

Write-only transactions are a group of write operations that appear to all take effect at the same time. They can also span data spread across many different shards. Write-only transactions are fully isolated from other write-only transactions and read-only transactions. Thus, a read-only transaction will either see all or none of a write-only transaction.

Partial Replication. Large-scale web services like Google and Facebook are typically deployed across many datacenters, e.g., all 9 in Figure 1. Deploying across so many datacenters necessitates partially replicating data to only a subset of the datacenters [7], [20]. K2 is designed to provide lower latency for such large-scale services for whom partial replication is necessary. We are also motivated to provide lower latency for the much larger number of medium-scale services for whom partial replication over many datacenters is a deployment option and not a requirement.



(a) User to far-away FE that is near BE with all the data.

(b) User to nearby FE that contacts the far-away BE twice to ensure consistency.

(c) User to nearby FE that goes to far-away BE at most once. (K2's worst case.)

(d) User to nearby FE that can access all data locally. (K2's common case.)

Fig. 2: User latency for requests to frontend web servers (FE) that access backend storage (BE).

B. Partial Replication for the Many

Using many datacenters is not required for medium-scale web services. Sufficient resources are available on cloud platforms to store all data and handle all user requests. These services can place all frontend and backends in 1 datacenter, e.g., on the West Coast (A). While West Coast users would get low latency, those elsewhere like Japan would have high latency from connecting to a far-away datacenter. A better option is to place frontends and backends in 3 geographically dispersed datacenters, e.g., West Coast (A), Europe (B), and Japan (C). This reduces user latency by serving requests closer to them. And, because data is typically replicated 3× for fault tolerance, moving from 1 to 3 datacenters has little effect on the cost of the deployment. Yet, it leaves many users still connecting to far-away datacenters, e.g., Australian users. Figure 2a shows such a connection to a far-away frontend.

One option would be to place frontends and fully replicated backends in a large number of datacenters. Unfortunately, this option is expensive. For instance, moving from 3 to 9 datacenters would roughly triple costs as the 6 additional replicas of data are not necessary for fault tolerance.

An appealing alternative is to deploy a service with frontends and partially replicated storage with 1/3 of the data in each datacenter. The cost for such a deployment would be roughly the same as using 3 datacenters with full replication. Such a deployment, however, can result in *higher* latency for end users if storage servers in the nearby datacenter contact far-away datacenters multiple times. This is not a concern for storage systems that provide eventual consistency because any data can be returned with any other data.

In storage systems that provide stronger guarantees—e.g., causal consistency—multiple round trips to far-away datacenters would be common. For instance, consider deploying a storage system in a configuration we call *replicas across datacenters* (RAD) where 1/3 of each replica is placed in each of the 9 datacenters. In this setting, the COPS [38] and Eiger [39] systems would require as many 2 and 3 sequential round-trips to far-away datacenters respectively. In COPS and Eiger, a first round optimistically reads data, and a second round is required if the data returned in the first round is inconsistent. Eiger incurs an additional delay of one round-trip between datacenters to check the status of pending updates if the requested data is being modified by ongoing transactions. Even 2 round-trips result in higher latency than a deployment with full replicas in 3 datacenters as shown in Figure 2b. Avoiding such scenarios motivates our first design goal.

Design Goal 1: At Most One Round of Non-Blocking Cross-Datacenter Requests. When a partially-replicated storage system needs at most one round of cross-datacenter requests that do not block its latency will at worst be similar to having full replicas in 3 datacenters, as shown in Figure 2c.

Design Goal 2: Often Zero Cross-Datacenter Requests. Achieving design goal 1 gives K2 the best possible worst-case end-user latency, matching full replication. But it is also no better. To provide a latency benefit K2 must avoid cross-datacenter requests in the common case. Such a scenario is shown in Figure 2d. This scenario is possible in a RAD deployments. However, it is unlikely as it requires all the data needed to serve a user's request to be in the 1/3 of data located in the nearby datacenter. Our design goal 2 is thus to *often* complete with zero rounds of cross-datacenter requests.

III. K2 Basic Design

This section presents the basic architecture of K2. Sections IV presents the replication design in K2. Section V completes the design with our read-only transaction algorithm.

Figure 3 shows the architecture of K2. We base our design on fully-replicated Eiger [39]. K2 inherits the mechanisms for tracking and enforcing causal consistency, local write-only transactions, and garbage collection from Eiger. The major changes in our design include our new algorithms for replication, cache-aware read-only transactions, and an LRU-like cache replacement policy. We also introduce some changes to Eiger's replicated write-only transaction algorithm to achieve our design goal 1.

A. Server Side Design

Within each datacenter, the keyspace is *sharded* across servers that are each responsible for a subset of the keyspace. For simplicity, our discussion here focuses on the simple key-value storage model, though our implementation uses the richer column-family data model [18], [34].

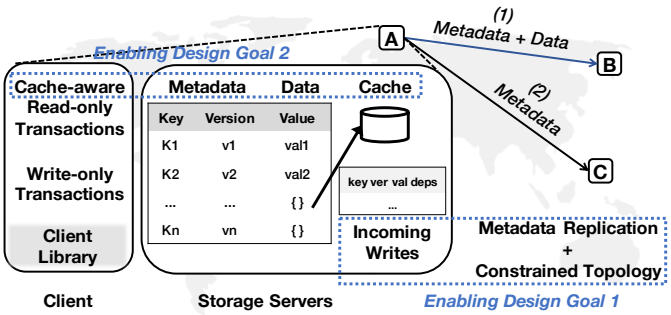


Fig. 3: The system architecture of K2. Each datacenter has the entire keyspace sharded across servers, and a small amount of cache. It stores data for a subset of keys and only metadata for the rest. K2 partially replicates data and fully replicates metadata by following a constrained topology.

Data and Metadata. Each datacenter stores the metadata for the entire keyspace and *data* (i.e., *values*) for a subset of keys. *Metadata* includes a *key* and a *version number*, which uniquely identifies the value of this key and is assigned by the datacenter that wrote this version. If a datacenter D always stores the value of a key K, then D is a *replica datacenter* of K, and K is a *replica key* in D. If D does not always store the value of K, then D is a *non-replica datacenter* of K, and K is a *non-replica key* in D. The value for each key is stored in a set of f datacenters, which tolerates up to $f-1$ simultaneous datacenter failures. We assume the mapping of keys to their f replica datacenters is known to each datacenter.

Cache. K2 augments each server with a small amount of cache containing additional values. K2 uses its cache to help achieve our design goal 2 of often avoiding any cross-datacenter requests. K2 caches a value of a non-replica key after fetching it from remote datacenters. K2 also temporarily caches the values for the writes of non-replica keys from local clients. An advantage of the cache in K2 is that writes of non-replica keys can commit locally and thus have low latency. K2's read-only transaction algorithm leverages the cached values to often avoid cross-datacenter requests and satisfy read-only transactions entirely in the local datacenter. We implement an LRU-like cache-eviction policy.

Clock. Servers and clients keep Lamport clocks [35], which advance upon message exchange. All operations are uniquely identified by a Lamport timestamp. The high-order bits of the timestamp are the Lamport clock, and the low-order bits are the unique identifier of the stamping machine.

B. Client Library

Each client has a *client library* that works in tandem with the storage servers. The library has two roles: First, it is the interface between the client and the storage system. Second, it tracks and attaches metadata to requests to help ensure that writes appear in a causally-consistent order.

The client library routes operations to the appropriate servers in the local datacenter and executes the read-only transaction and write-only transaction algorithms. It helps the storage systems ensure writes appear in a causally-consistent order using two types of metadata. The first is by attaching a unique *Lamport timestamp* to each write. Storage servers use these timestamps with a last-writer-wins policy [54] to ensure causally later writes always overwrite earlier ones [38]. The second type of metadata the client library tracks is *explicit causal dependencies*. The client library tracks only the *one-hop dependencies, deps,* that include the client's previous write and the writes of all values it has read since that write. Lamport timestamps combined with explicit one-hop dependencies are sufficient to ensure causal consistency while introducing less overhead than vector clocks [39].

Each dependency is a <*key, version*> pair. The client library updates its *deps* after a read-only or write-only transaction completes. It includes *deps* when it sends write-only transactions to the local datacenter. The dependencies are then used during inter-datacenter replication to enforce causal consistency. Using one-hop dependencies frees servers from needing to store metadata about causal dependencies and is sufficient for enforcing causal consistency [39].

C. Local Write-Only Transactions

In K2, a client commits its write-only transactions in its local datacenter by following a variant of the two-phase commit protocol [50]. The client splits keys into sub-requests and sends each to the corresponding servers—i.e., *participants*—in the local datacenter. It picks one key at random to be the *coordinator-key*. The participant holding this key is the *coordinator* and the others are *cohorts*. Each participant prepares by marking the keys in its sub-request as pending and sends *Yes* to the coordinator. Once all cohorts prepare, the coordinator assigns its current logical time (Lamport timestamp) as the *version number* and the *earliest valid time (EVT)* of this transaction. The version number uniquely identifies this transaction, and is used in other datacenters to apply writes in the correct causal order. The EVT indicates the logical time when the new versions are made visible in a datacenter, and is used locally in this datacenter as part of the read-only transaction protocol. The coordinator commits the transaction, sends each cohort a *Commit* that includes the version number and EVT. It then replies to the client with the version number. If writing to a non-replica key, the server commits only the metadata—i.e., the key, version, and EVT—and caches the value. Caching the write reduces read latency by avoiding unnecessary remote fetches for this value. The client library updates its *deps* and the read timestamp (§V) to maintain causal consistency. It updates *deps* by first clearing it and then adding the <*coordinator-key, version*> pair to *deps*.

IV. Replication Design

This section presents K2's replication design (§IV-A), and how its design decisions progressively build on each other to provide at most one round of non-blocking cross-datacenter read requests (§IV-B).

A. Replication Design

Metadata Replication and Constrained Replication Topology. *Metadata replication* decouples data and metadata replication. It replicates the *metadata* of a write—i.e., key, version, and dependencies—to all datacenters, and replicates the *data* of a write—the value—only to replica datacenters. *Constrained replication topology* orders the replication to replica datacenters before the replication to non-replica datacenters.

Replication of Write-Only Transactions. Replication of a write-only transaction is done by each participant (coordinator/cohort) in its local datacenter after it commits locally (§III-C). Each participant *asynchronously* replicates each key in its sub-request to its *equivalent* participants—the servers storing the same key—in other datacenters in two phases. In the first phase, the local participant replicates data and metadata to the replica participants in parallel. When a participant receives a replicated write that includes data, it immediately stores it in the *IncomingWrites* table before sending an acknowledgment to the sender. Once the local participant has been notified by all replica participants, it proceeds to the second phase. In the second phase, it replicates the metadata and the list of replicas storing the value to the non-replica participants. Only the coordinator needs to include causal dependencies with its metadata replication because each remote coordinator does dependency checks for its transaction group. K2's replication is asynchronous and *not* on-path for client-facing operations. Hence, it does not affect the latency of any client's operations. K2 introduces the IncomingWrites table to make the new data accessible *only* to remote reads while the transaction is pending. This table is *not* visible to local reads.

Committing Replicated Write-Only Transactions. Replicated write-only transactions are committed using a protocol that is a variant of two-phase commit. Each cohort notifies its transaction coordinator after receiving the replicated sub-request of the transaction. Concurrently, the coordinator issues the dependency checks for the transaction by contacting the local servers responsible for those dependencies. A local server replies to the dependency check immediately if the specified $<$*key, version*$>$ is committed, otherwise it waits until it is committed to reply. The coordinator then waits for all dependencies to verify and to be notified by all cohorts before beginning two-phase commit. This waiting for one-hop dependencies before applying replicated writes provides causal consistency [39]. The coordinator sends each cohort a *Prepare*. Once all cohorts reply, it sets this transaction's EVT to its current logical time, commits the transaction, and sends each cohort a *Commit* that includes EVT. Each participant deletes this transaction's sub-request from the IncomingWrites table after it commits the transaction.

Multiversioning Framework and Applying Replicated Writes. K2 keeps multiple versions of a key for a short time. Multiversioning enables K2's efficient read-only transaction algorithm. How a server applies a write depends on the current version it has for a specified key and if it is storing the data. When applying a write a server compares its version number with the version number of its most recent write to the same key. The version numbers are assigned by the datacenters that accept the writes based on Lamport timestamps and are consistent with the causal ordering of writes. Thus, a server should only make the write visible to local reads if its version number is greater than its most recent write. For non-replica servers, this results in them either applying the write if it is newer than the current value or discarding it entirely if it is not. This procedure would not be safe for replica servers, however, because the write might be needed to serve a remote read. Replica servers thus apply the write in all cases, store it in the multiversioning framework, and make it available *only to remote reads* if it is older than the current value.

Garbage Collection (GC). K2 keeps a version around if it is not older than 5 s, or this version or any of its earlier versions has been accessed by the first round of a read-only transaction within the past 5 s, the configurable transaction timeout. K2 performs garbage collection lazily whenever a new version of a key is inserted and then removes any old versions that do not satisfy either of the two conditions. GC is a common component in multiversioning data stores to keep memory and storage footprints low [38], [39], [51]. K2's GC is similar to Eiger's [39] with the addition of keeping around all versions not older than 5 s to enable our cache-aware read-only transaction algorithm.

B. Rationale and Key Insights

K2's replication design differs significantly from past work and is what ensures at most one round of non-blocking cross-datacenter requests. At the lowest level is K2's metadata replication design that ensures at most one round of cross-datacenter requests. Above that K2 layers a constrained replication topology and a write-only transaction algorithm that together ensure cross-datacenter requests do not block.

Metadata Replication. Partial replication of the data gives K2 most of the storage capacity benefit of a partially-replicated storage system, while full replication of the metadata enables K2 to achieve at most one round of cross-datacenter requests. The key insight is that metadata is all that is necessary to determine what data a client can consistently read. K2 fully replicates metadata, consistently updates metadata in each datacenter, and then runs its read-only transaction algorithm on that consistent metadata in the local datacenter to determine consistent data versions. Then, only a single round of cross-datacenter read requests is required if the consistent versions are not stored locally. K2 can thus avoid multiple unnecessary rounds of cross-datacenter requests to figure out consistent data values to read.

Decoupling data and metadata replication, however, introduces a new challenge that can lead to blocking. The metadata replication in a non-replica datacenter can race ahead of data replication in replica datacenter. Then, when the non-replica datacenter requests a specific value from the replica datacenter its request will need to block until that value arrives. K2 overcomes this challenge to ensure cross-datacenter requests

do not block with its constrained replication topology and write-only transaction algorithm.

Constrained Replication Topology. K2's constrained replication carefully orders how data and metadata are replicated to replica and non-replica datacenters to ensure a datacenter always knows where to read a value without blocking. This ordering provides an important invariant: once a non-replica datacenter learns about an update, the value must be available from each of the replica datacenters.

This invariant is sufficient to ensure cross-datacenter requests do not block for writes to individual keys. It, however, breaks existing algorithms for write transactions that atomically update multiple keys. These existing algorithms include general transaction algorithms like two-phase locking and optimistic concurrency control as well as specialized write-only transaction algorithms like Eiger's [39]. The existing algorithms break because they include two-phase commit, which waits for all participants in a transaction to prepare successfully before any commit. For example, consider a write transaction that updates keys A and B that are replicated in disjoint datacenters. Using the invariant, the replica datacenters for key A will not be able to prepare non-replica key B until they know it has committed in its replica datacenters and thus is available for reads. But the same is true for the replica datacenters for key B, they will not be able to prepare non-replica key A until they know it has committed. Thus, the different sets of replicas are deadlocked and never commit. K2 sidesteps this issue with its write-only transaction algorithm.

Replicated Write-Only Transactions. The key insight behind K2's write-only transaction algorithm is to decouple the availability of data for remote reads from its availability for local reads. Data should be available for remote reads immediately and for as long as necessary to ensure remote reads can be served without blocking. While data should be available for local reads only when it satisfies the guarantees of the storage system. This decoupling allows K2 to provide its invariant that ensures remote reads do not block. It breaks down into two cases: before and after a replica datacenter applies a write to make it visible to local reads. Before a write is applied, K2 makes it *available only to remote reads* through the IncomingWrites table. This is safe since K2 ensures that the remote read only requests a version that is already causally consistent in the requesting datacenter. After a write is applied, K2 keeps it in the multiversioning framework until it can be safely garbage collected.

V. Read-Only Transactions

This section completes K2's design by describing our cache-aware, read-only transaction algorithm. The algorithm is built around two key insights that allow it to often avoid any cross-datacenter requests: cache awareness and trading a little freshness for a lot of performance.

A. Cache Awareness

The read-only transaction algorithm exploits the temporal locality of data access by leveraging the data cached as part of K2's design. Caching values which are likely to be accessed again soon [5], [9], [17], [28], [48] avoids unnecessary remote fetches of the same data. For instance, after Alice uploads a new photo (cache after write), she is likely to verify the upload was successful by downloading it (read the cached photo). Similarly, after Bob reads a photo (cache after remote fetch), the same photo will likely be recommended to Bob's friends (read the cached photo). The benefits of caching are even more promising for real-world applications, which usually exhibit Zipfian workloads, i.e., most operations are on a small subset of the data. For instance, Facebook's TAO caching system reported an overall hit rate of 96.4% [16].

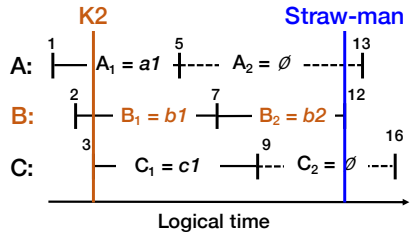


Fig. 4: A and C are non-replica keys, B is a replica key. a_1 and c_1 are cached versions. A straw-man solution incurs unnecessary remote fetches, while K2's read-only transaction reuses cached versions when safe.

Caching makes it possible to avoid many cross-datacenter requests. The challenge, however, is realizing this possibility with our read-only transaction algorithm. Previous algorithms focused on providing low latency and consistency. Our algorithm adds the need to reuse cached values.

B. Trading Freshness for Performance

K2's read-only transactions provide causal consistency. Causal consistency has two properties we can leverage to achieve better read performance. First, it does not require a read to reflect the most recent updates, commonly known as the real-time requirement in stronger consistency models, e.g., linearizability [30]. Second, it does not require all clients to advance their views of the system at the same rate: it is *technically* causally consistent if the system keeps making a client read at a fixed timestamp that only advances when the client issues writes. Our algorithm does much better than this as we guarantee that clients make progress through the garbage collection that safely discards any versions older than 5 s. In addition, in our evaluation we find much lower staleness with a median of no staleness at all.

With these two observations in mind, we explore the possibility of avoiding remote fetches by allowing each client to maintain and manage its read timestamp, which could be slightly stale but for which most non-replica items have cached values. For instance, in Figure 4, a straw-man solution for read-only transactions is to read at the most-recent timestamp, 12. However, this will incur two unnecessary remote fetches on A_2 and C_2 since those versions are not present locally. Instead, K2's read-only transaction algorithm reads at timestamp 3 since both A and C have cached values at that timestamp.

```
1  function read_txn(<keys>):
2    vers[][] = [][], vals[] = []
3    for k in keys: /* 1st round */
4      vers[k] = read(k, cli.read_ts)
5    ts = find_ts(vers)
6    for k in keys:
7      for ver in vers[k]:
8        if ver.evt ≤ ts ≤ ver.lvt and
9           ver.val != null:
10         vals[k] = ver.val; break
11     if !vals.contains(k):  /* 2nd round */
12       vals[k] = read_by_time(k, ts)
13   cli.read_ts = max(cli.read_ts, ts)
14   for k in keys: deps.add(k, vals[k].ver)
15   return vals
```

Fig. 5: Pseudocode for read-only transaction.

C. Read-only Transaction Algorithm

Figure 5 shows the pseudocode for K2's read-only transaction algorithm at a client. Each client maintains a *read timestamp* `read_ts`, and includes this timestamp when it sends read-only transaction requests to the servers. The client begins with a round of parallel requests to the servers in its local datacenter. Each server returns all visible versions of each key in its request that are valid at or after `read_ts`. Each version includes the version number, EVT, LVT, and value if it is stored or cached locally. The *LVT (latest valid time)* of a version is the latest logical time before it is overwritten by a new version. The server returns its current logical time for LVT if the version is the latest. The server returns an empty value if the version or any of its earlier versions are pending. (The empty value indicates that the a version is potentially being modified by some ongoing write-only transactions.) The client examines the returned versions and finds a consistent logical time `ts` that minimizes cross-datacenter requests. Specifically, `find_ts` examines the EVTs of all returned versions. It finds the earliest EVT where either (1) all keys have a valid value, or (2) all non-replica keys have a valid value, or (3) the most keys have a valid value. This procedure for picking the effective logical time is what makes our algorithm cache-aware.

A second round of read requests is required if a key has no consistent version or value at `ts`. If the key is being modified by pending write-only transactions earlier than `ts`, the server waits for the pending transactions to commit. This waiting does not appreciably affect latency because the longest a write-only transaction will remain pending is a single roundtrip within the *local* datacenter (from the cohorts to the coordinator and back). Once the pending transactions commit, the server determines the committed version at time `ts`, and returns the value if it is available. If not, the server sends a remote read request to its equivalent server in the nearest replica datacenter to fetch the value given the key and the version number. Our constrained replication topology and write-only transaction algorithm ensure the requested version will be accessible in the replica datacenter. The remote server checks its IncomingWrites table and multiversioning framework for the requested version, and sends its value to the requesting server. Upon receiving the response, the local server caches the value and replies to the client. To maintain causal consistency, the client updates its `read_ts` and dependencies. It also advances `read_ts` to *max*(`read_ts`, `write_ts`) after it completes a local write-only transaction that returns a write timestamp `write_ts` (§III-C).

VI. Fault Tolerance

This section describes unimplemented extensions to K2 for handling failures and enabling clients to switch datacenters. These extensions are similar to prior work [38], [39].

A. Handling Failures

Server failures within a DC: Server failures are unavoidable in practice. K2 can provide availability for a logical server despite failures using a fault-tolerant protocol like Paxos [36] or Chain Replication [55].

Datacenter failures: With a replication factor of f, K2 assumes up to $f-1$ replica datacenters can fail. Replicating writes to replica datacenters can proceed if at least one replica datacenter of each key in those writes is available. The non-replica datacenters can send their remote read requests to the available replica datacenters. Permanent datacenter failures (e.g., a datacenter being destroyed by a tsunami) may lead to data loss in K2 if a local datacenter is destroyed after replying to a client's write request but before successfully replicating them to any other datacenter. This cost of achieving low latency for local writes that return faster than inter-datacenter latency is inevitable [38], [39]. Transient failures (e.g., temporary power failures) do not result in data loss. However, the local (temporarily failed) datacenter should replicate its pending updates to other datacenters once it is restored.

B. Switching Datacenters

The clients of K2 are frontend servers co-located in the same datacenters as the backend storage servers of K2 (§II-A). These clients will continue to access their co-located servers. The users they issue operations on behalf of, however, may wish to switch datacenters, e.g., after flying to another part of the world. K2 can allow users to switch datacenters using the following steps: (0) Dependencies are propagated back to users, e.g., in an HTTP cookie. (1) When a user switches to a new datacenter it sends its dependencies to its frontend, e.g., the user request includes the cookie. (2) That frontend checks (by polling with reads) and waits until all dependencies (which includes its last write and all its reads since the last write) are satisfied by the metadata in the local datacenter. (3) That frontend then uses the included dependencies for this user. Steps 0 and 1 ensure the new frontend knows the dependencies for this user. Step 2 ensures all causal dependencies are present in the new datacenter. Step 3 ensures later operations on behalf of this user include the correct dependencies.

VII. Evaluation

Our evaluation compares K2 to RAD, a baseline that directly adapts causal consistency for partial replication, and

	VA	CA	SP	LDN	TYO
CA	60				
SP	146	194			
LDN	76	136	214		
TYO	162	110	269	233	
SG	243	178	333	163	68

Fig. 6: Round trip latencies in ms between datacenters emulated on Emulab and based on EC2 measurements.

PaRiS*, a baseline that uses a per-client cache [51], to understand the improvements and tradeoffs of K2's design. Specifically, our evaluation answers these questions:

§VII-C What improvement in latency does K2 provide?

§VII-D How does the throughput and write latency of K2 compare to the RAD baseline?

§VII-D What staleness does K2's new read-only transaction algorithm introduce?

A. Implementation and Baseline

K2 is implemented as a modification to the Java codebase of Eiger, a scalable geo-replicated storage system that provides causal consistency [39]. The major changes in our implementation include our new algorithms for replication, write-only transactions, read-only transactions, LRU-like cache replacement policy, and garbage collection.

Replicas Across Datacenters (RAD). We use a direct adaptation of scalable causal consistency to partial replication as our baseline for comparison. We compare to RAD because it is a reasonable adaptation of a fully-replicated causal consistency design to a partially replicated setting. To implement RAD, we configure Eiger to split data in each replica across datacenters, which together form a replica group. Clients send read and write requests directly to the datacenters in its group that hold the relevant keys. A datacenter in a group needs to replicate writes to its equivalent datacenters, which hold the same key ranges, in other groups. Before committing a replicated write, a datacenter sends dependency checks to other datacenters in its group. It applies the replicated write once all dependencies are satisfied. RAD uses Eiger's read-only and write-only transaction algorithms.

It is not straightforward to adapt the design of Eiger to make efficient use of a cache. Eiger's read-only transaction algorithm's first round returns the currently visible value for each key within a replica. A local datacenter cache would only contain previously read values and would not know if these values were still visible. All first round requests for non-replica keys would thus need to contact a remote datacenter. This precludes the possibility of achieving zero cross-datacenter requests, which is the purpose of our cache. Thus, our RAD baseline does not include a datacenter cache.

PaRiS*. We also implement another baseline, PaRiS*, which uses a per-client cache and has at most one round of reads [51]. PaRiS* implements a subset of the full design of PaRiS and provides slightly optimistic lower-bounds on the latency of a full PaRiS implementation. We modify K2's implementation to augment each client with a private cache as in PaRiS. A client's recent writes are kept in its cache for 5 s. This is longer than they would be in cache for a full PaRiS implementation which will clear them once their timestamp is passed by the Universal Stable Time. PaRiS*'s read-only transactions take at most one round of non-blocking remote reads as in PaRiS.

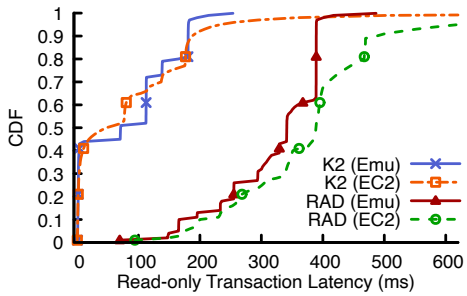


Fig. 7: Comparing K2 and RAD on EC2 and Emulab with the default workload. Results are similar with K2 providing a significant improvement at all percentiles: its average latency improvement is 297 ms on EC2 and 243 ms on Emulab.

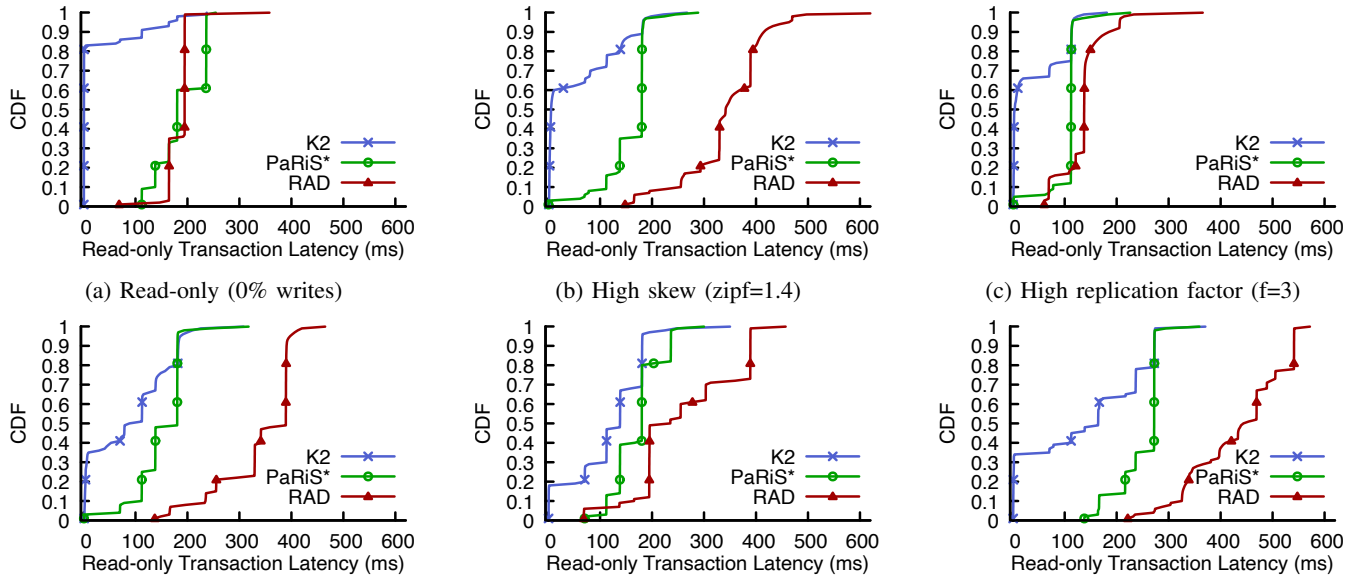
B. Experimental Setup

Most experiments are run on the Emulab testbed [57] where we have exclusive bare-metal access to 72 machines. Each machine has one 2.4 GHz 64-bit 8-Core E5-2630 "Haswell" processor, 64 GB 2133 MT/s DDR4 RAM, and are networked with 1 Gbps Ethernet. Emulab machines are physically co-located so we emulate the latency between datacenters. We validate this use of emulated latency on Emulab by running some experiments on Amazon EC2 in geo-distributed regions. On EC2 we use t3.2xlarge instances, which have CPU and memory specifications comparable to the machines on Emulab testbed: each has 8 virtual CPUs and 32 GB of memory.

Configuration and Workloads. We use 72 machines configured as 6 datacenters with 4 servers and 8 co-located clients in each. Machines in the Emulab testbed are physically co-located, so we use Linux's tc to emulate wide-area latency between datacenters. To emulate a globally-distributed deployment, we choose locations that are spread around the world: Virginia (VA), California (CA), São Paulo (SP), London (LDN), Tokyo (TYO), and Singapore (SG). The wide-area latencies are based on latencies between EC2 regions [11].

Each set of clients reads from and writes to their local datacenter. We measure system throughput as the total throughput of all datacenters. We generate the workload using Eiger's benchmarking system with SNOW's [40] addition of Zipf request generation. All experiments use 1 million keys, 128 byte values, 5 keys per operation, and 5 columns per key. Unless otherwise specified all experiments use a cache size of 5% of the total keys, a Zipf constant of 1.2, a write percentage of 1%, a write-only transaction percentage of 50% (of writes), and a replication factor of 2. We experimentally vary each of these parameters to observe their effect.

Most experiments use a write percentage of 1% because most workloads are read heavy. Our choice of 1% is a compromise between the 0.2% writes reported for Facebook's



(a) Read-only (0% writes) (b) High skew (zipf=1.4) (c) High replication factor (f=3)

(d) High write percentage (5%) (e) Moderate skew (zipf=0.9) (f) Low replication factor (f=1)

Fig. 8: Read-only transaction latency. K2 provides significantly lower latency than PaRiS⋆ and RAD at all percentiles for all tested workloads. The average improvement of K2 over PaRiS⋆ and RAD is 53–165 ms and 88–297 ms, respectively.

production TAO system [16], and the 5% writes in YCSB's workload B [19]. We also evaluate with other write percentages that match realistic workloads: 0.0% (YCSB's workload C [19]), 0.1% (approximate write percentage reported for Google's advertising backend F1 on Spanner [20]), 0.2% (Facebook's production TAO system), 5% (YCSB's workload B). Most experiments use a Zipf constant of 1.2 because most workloads are highly skewed. We are not aware of specific skew numbers for storage systems like K2, so we based skew on the reported access characteristics of Facebook photos [31]. Facebook photos were reported to follow a power-law distribution with $\alpha = 1.84$, which is equivalent to a Zipf constant of 1.2. We test with Zipf constants as low as 0.9 and high as 1.4. A Zipf constant of 1.4 is equivalent to the $\alpha = 1.72$ power-law distribution observed for Facebook videos [53].

Methodology. To fairly configure our later experiments we probed the operation of each system under increasing load. For each system in the latency experiments, we choose the number of closed-loop client threads on each of the 48 client machines where the system operates at medium load. This is in the appropriate range for production systems [56] and reduces the effect of queuing delays. Each data point we report is the median of 3 trials that each last for 12 minutes. This duration is sufficiently long to warm up the cache, i.e., most keys are requested at least once. We omit the first 9 minutes and the last 20 seconds of each trial to exclude the cache warm-up period and experimental artifacts.

Validating results on Amazon EC2. We deploy K2 and RAD on Amazon EC2 datacenters in the six different locations with actual wide-area latency shown in Figure 6. Network bandwidth is not the bottleneck in our evaluation settings. K2's write-only transaction latency is low on both EC2 and Emulab since K2 commits writes locally and its commit is not affected by the network delay. Figure 7 shows the CDFs of read-only transaction latency under default settings. There are three differences in the results. First, EC2 results are smoother due to slight variations in actual latency and noise from the virtualized environment. Second, the EC2 results have a longer tail: the 99.9th percentile latency is ~1 second for K2 and ~1.4 seconds for RAD. Third, the latency improvement of K2 is higher on EC2 than it is on Emulab: the average latency improvement on EC2 is 297 ms and 243 ms on Emulab. We observe that the distributions and trends are similar on Emulab with emulated latency and on EC2 with actual wide-area delays. We are thus confident that results from Emulab with emulated latency are indicative of deployments on cloud platforms. If there is any appreciable difference, it is that K2 latency improvement would be *greater* in a deployment on a cloud platform. We run experiments on Emulab for higher repeatability and lower cost.

C. Read Latency Improvement

K2 is designed to decrease the latency of read-only transactions over partially replicated storage. Figure 8 shows the latency of read-only transactions in K2, RAD, and PaRiS⋆ under a variety of workloads. We find that K2 significantly improves latency compared to RAD and PaRiS⋆ at all percentiles in all evaluated workloads. The magnitude of these improvements varies with the workload. In most workloads, K2 provides an average latency improvement of 140–297 ms over RAD, and 53–165 ms over PaRiS⋆. This significant latency improvement of K2 over the baselines is enabled by K2's novel design which optimizes latency of read-only transactions by providing all-

	Default	Replication		Write (%)		Zipf		Cache (%)	
		f=1	f=3	0.1	5	0.9	1.4	1	15
K2	41.6	21.1	53.7	47.7	26.0	21.3	46.3	30.9	44.3
RAD	24.8	11.7	51.9	59.0	20.2	85.4	14.8	24.8	24.8

Fig. 9: Throughput (K txns/sec) under different settings.

local latency more often and guaranteeing the best possible worst-case latency.

More All-Local Latency. RAD does not cache non-replica data in datacenters, so any read-only transaction that accesses non-replica data must go to a remote datacenter. This happens >99% of the time in all workloads as shown by RAD's 1st percentile latency being >60 ms, the lowest inter-datacenter latency. PaRiS* uses a per-client cache to keep client's recent writes. PaRiS* provides local latency when all requested keys are replica keys or are stored in the client's private cache. This happens <6% of the time in all workloads as shown by PaRiS*'s 6th percentile latency being >60 ms. >95% of PaRiS*'s read-only transactions must contact a remote datacenter and thus incur high latency. K2 caches a small percentage of non-replica data in each datacenter and uses them when safe. This allows K2 to serve many read-only transactions entirely locally.

K2 provides local latency for 19–83% of read-only transactions depending on the workload. K2's most significant improvements are for the highly skewed workload (8b), the high replication factor (8c), and the read-only workload (8a). Its smallest improvement comes with a moderately skewed workload (8e). This is as expected because more skewed workloads are easier to cache. The percentage of transactions with all-local latency decreases with a higher write percentage (8d) and with a lower replication factor (8f). These changes are also due to changes in the effectiveness of the cache. For instance, increasing the replication factor from 2 to 3 results in 33% less non-replica data vying for a spot in the cache.

Best Possible Worst-Case Latency. K2 and PaRiS* achieve the best possible worst-case latency for read-only transactions on partially-replicated data: they need at most one inter-datacenter round trip to fetch the values of non-replica keys. In contrast, RAD needs two inter-datacenter round-trips if the non-replica keys fetched in the first round of the read-only transaction are not consistent. Figures 8b, 8d, and 8f show workloads where RAD issues the second round of remote reads often: high skew, high write percentage, and low replication factor. In each of these workloads 91–98% of read-only transactions take two wide-area rounds.

Facebook TAO Workload. We experiment with a synthetic workload that uses the value sizes, columns/key, and keys/operations reported for Facebook's TAO system [16], [39]. We use the default Zipf constant of 1.2 since it is not reported in TAO. We find that K2 provides local latency for 73% of read-only transactions, while PaRiS* and RAD achieve local latency for <1% of read-only transactions.

D. Throughput, Write Latency, Staleness

Throughput Comparison. K2 aims to avoid remote reads by leveraging cached values and thus can potentially improve throughput by reducing the number of requests in the system. However, K2 has three sources of overhead: replicating metadata to non-replicas, doing dependency checks before applying replicated metadata, and returning multiple versions in its read-only transaction algorithm. We quantify the throughput overhead of K2 compared to RAD.

Figure 9 shows the peak throughput of the systems for several settings using the minimum and maximum values of each parameter while keeping the others at their default. We observe that in many settings (e.g., high write percentage of 5%, and highly skewed Zipf constant of 1.4), K2 provides higher throughput than RAD. Under these workloads, RAD often needs the second round of reads to request consistent versions of the contended keys and is bottlenecked by a small set of servers. K2 avoids the bottleneck better by allowing each datacenter to read a slightly older, consistent version of highly contended keys from its local cache, and thus avoids imposing high remote read loads on the replica datacenters of those keys. In some settings (e.g., a moderately skewed Zipf constant of 0.9), we find that RAD has higher throughput than K2. Unlike K2, each datacenter in RAD handles dependency checks and replication for only replica keys, leaving more CPU and memory capacity to serve local client requests.

Write Latency. K2 achieves much lower write latency than RAD for single-key writes and write-only transactions because K2 can commit write operations locally, while RAD often must contact remote datacenters. For instance, under our default settings K2's 99th percentile latency is 23 ms for write-only transactions while RAD's 50th percentile latency is 147 ms for simple writes and 201 ms for write-only transactions.

Data Staleness. K2 aims to satisfy read requests entirely inside a local datacenter by leveraging older cached versions and thus accepts some staleness for better performance. Staleness is measured on servers as the time since a newer version of that key has been written. For instance, if the returned version is the newest version on the server, the staleness is 0. Or, if the returned version was overwritten by a newer version 100 ms ago, the staleness is 100 ms. RAD provide 0 staleness if its read-only transactions complete in one round. We quantified the staleness in K2 for write percentages between 0.1–5%. The median staleness is 0 ms for all cases, 75th percentile staleness is 105 ms or less, and 99th percentile staleness is between 516 and 1117 ms. We expect this staleness to be an acceptable tradeoff for the lower latency provided by K2.

VIII. Related Work

This section reviews previous partially replicated systems, fully replicated systems, and systems that provide causal consistency. K2 is primarily distinguished by being the first work to realize the low latency benefit of many datacenters with strong guarantees: causal consistency, read-only transactions, and write-only transactions.

Partially-Replicated Data Stores. PRACTI [12] is a classical partial replication system that supports topology independence, i.e., any-to-any replica propagation, and provides arbitrary

consistency. K2 builds on PRACTI's insight to separate the control path (metadata) and the data path (data replication). One difference in K2 is our use of a cache—and our algorithms that exploit it for many clients—in each datacenter, and the constrained replication topology to provide non-blocking remote reads. More importantly, PRACTI was designed for a different era when all data that would be accessed together could fit on a single machine. Hence, its design is based on exchanging logs of serialized updates and is not scalable, i.e., it is designed for at most one shard in what are now datacenters.

Karma [41] is concurrent work on enabling a causally-consistent data store to support partial replication that uses an approach similar to the replicas across datacenters baseline we compare to in our evaluation. It focuses on allowing clients to switch the datacenter to which they are connected and enabling simple reads from a cache in a datacenter; it does not support write-only transactions or read-only transactions. K2, in contrast, does not focus on allowing clients to switch datacenters though it could be extended to do so (§VI-B). K2 focuses on providing write-only and read-only transactions.

Spanner [20] is Google's globally-distributed data store, which provides strict serializability and partial replication. K2 targets a much lower latency setting than Spanner with guarantees that are compatible with handling all reads inside the local datacenter as well as trading away some capacity for significant read latency improvements from caching.

PaRiS [51] is a concurrently developed causally-consistent data store that supports partial replication. PaRiS uses per-client private caches and a universal stable time (UST) to provide causal read-write transactions, which are stronger than K2's guarantees. K2's guarantees are, however, still useful for a large set of applications (§II-A). PaRiS provides at most one round of non-blocking cross-datacenter requests like K2. PaRiS handles read-only transactions locally only when all requested keys are either replicated in this datacenter or are stored in the client's private cache because the client has written to them since the UST. As our experimental comparison with PaRiS* shows, this occurs rarely. In contrast, K2 is able to often handle read-only transactions locally. Similarly, PaRiS requires write transactions to contact remote datacenters except when all keys are replicated in this datacenter. In contrast, K2's write-only transactions always commit to the local cache.

PaRiS's use of per-client private caches and K2's per-datacenter shared caches are quite different. PaRiS's per-client cache is necessary for correctness. They cannot be shared between clients because they contain newer-than-UST state and thus it would be unsafe for one client to read data from another client's private cache. K2's per-datacenter shared caches, in contrast, are safely shared between all clients in a datacenter and enable K2 to often handle read-only transactions locally.

Improving Partial Replication. Volley [1], Tuba [8], and Akkio [7] deal with data placement and migration for partially replicated systems. They optimize data placement policies and dynamically migrate data to different replicas based on system logs to satisfy user requirements and reduce operational costs. This line of work is orthogonal to K2, which operates independently of any placement policy. Integrating such policies into K2 could further reduce latency by increasing the likelihood that a read's local datacenter is a replica datacenter.

Saturn [14] and C^3 [24] focus on improving the throughput and data visibility latency of a partially-replicated data store through novel metadata propagation and causal-consistency enforcing algorithms. Saturn and C^3, however, only support simple read and write operations. K2, in contrast, focuses on achieving lower latency for partial replication with stronger guarantees: read-only and write-only transactions.

Partial replication has also been studied in file systems [29], [42], [49]. This work focused on detecting and repairing conflicting updates [29], [42] or enabling good performance by aggressively creating new replicas. K2 instead focuses on higher-layer concerns like consistency and transactions.

Cache-Aware Read-Only Transactions. TxCache [47] uses a set of caches to increase the throughput of an underlying monolithic database while providing serializability within an application specified staleness bound. It uses a cache-aware read-only transaction algorithm that starts with a set of pinned snapshots identifiers from the underlying database and refines the set of acceptable identifiers as a transaction proceeds. TxCache's algorithm influenced our design, but it cannot be applied to our setting because we do not have a monolithic database that can pin snapshots of all the data. Instead, K2 determines if the local datacenter has cached values that can be used in a consistent snapshot dynamically.

Causal Consistency. Causal consistency is provided by many systems [2], [3], [12], [13], [24], [32], [33], [38], [39], [43], [46], [51], [52]. Excluding PRACTI [12], C^3 [24] and PaRiS [51], all these systems are built atop fully replicated data stores and inevitably suffer from its limitations. We based our design and implementation on fully-replicated Eiger [39]. Since Eiger, there have been many innovations [4], [14], [22]–[24] in reducing the granularity of metadata for tracking causal consistency and thus decreasing the throughput overhead of enforcing causal consistency in datacenters. These innovations are orthogonal to our contributions here; we believe it would be straightforward (though time-consuming) to incorporate these designs into K2 to achieve higher throughput.

IX. Conclusion

Deploying web services across many datacenters has the potential to significantly reduce end-user latency. Realizing this lower latency, however, is complicated by the need to partially replicate data in the backend storage system. K2 is a partially-replicated storage system that unlocks low latency for the strong guarantees of causal consistency, read-only transactions, and write-only transactions.

Acknowledgment

We thank our shepherd, José Orlando Pereira, and the anonymous reviewers for their helpful comments. We are grateful to Theano Stavrinos, Christopher Hodsdon, Jeffrey Helt, and Matthew Burke for their feedback. This work was supported by the National Science Foundation under grant number CNS-1827977.

REFERENCES

[1] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[2] Mustaque Ahamad, Gil Neiger, Prince Kohli, James Burns, and Phil Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[3] Deepthi D. Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguia, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, 2016.

[4] Sergio Almeida, Joao Leitao, and Luis Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *EuroSys*, 2013.

[5] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana De Oliveira. Characterizing reference locality in the WWW. In *PDIS*, 1996.

[6] https://aws.amazon.com/about-aws/global-infrastructure/, 2021.

[7] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with Akkio. In *OSDI*, 2018.

[8] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI*, 2014.

[9] Martin F Arlitt and Carey L Williamson. Web server workload characterization: The search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1), 1996.

[10] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[11] https://www.cloudping.co/, 2021.

[12] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *NSDI*, 2006.

[13] Kenneth P. Birman and Robbert V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[14] Manuel Bravo, Luis Rodrigues, and Peter Van Roy. Saturn: A distributed metadata service for causal consistency. In *EuroSys*, 2017.

[15] Eric Brewer. Towards robust distributed systems. PODC Keynote, July 2000.

[16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *ATC*, 2013.

[17] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USITS*, 1997.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2), 2008.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[21] https://engineering.fb.com/data-center-engineering/data-centers-2018/, 2021.

[22] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *SOCC*, 2013.

[23] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SOCC*, 2014.

[24] P. Fouto, J. Leito, and N. Preguia. Practical and fast causal consistent partial geo-replication. In *NCA*, 2018.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[26] https://cloud.google.com/about/locations/, 2021.

[27] https://www.google.com/about/datacenters/inside/locations/, 2021.

[28] Steven D Gribble and Eric A Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *USITS*, 1997.

[29] Richard Guy, John S. Heidemann, Wai Mak, Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *Summer USENIX Conference*, 1990.

[30] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 1990.

[31] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

[32] Diptanshu Kakwani and Rupesh Nasre. Orion: Time estimated causally consistent key-value store. In *PaPoC*, 2020.

[33] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[34] Avinash Lakshman and Prashant Malik. Cassandra – a decentralized structured storage system. In *LADIS*, 2009.

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

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

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

[38] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[40] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *OSDI*, 2016.

[41] Tariq Mahmood, Shankaranarayanan Puzhavakath Narayanan, Sanjay Rao, T. N. Vijaykumar, and Mithuna Thottethodi. Karma: Cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency. *IEEE TCC*, 2018.

[42] Dahlia Malkhi and Doug Terry. Concise version vectors in WinFS. In *DISC*, 2005.

[43] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! Scalable causal consistency with no slowdown cascades. In *NSDI*, 2017.

[44] https://azure.microsoft.com/en-us/global-infrastructure/regions/, 2021.

[45] Ruoming Pang, Ramón Cáceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, and et al. Zanzibar: Google's consistent, global authorization system. In *ATC*, 2019.

[46] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[47] Dan R.K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, 2010.

[48] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2), 2000.

[49] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI), 2002.

[50] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Info. Theory*, 9(3), 1983.

[51] K. Spirovska, D. Didona, and W. Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *ICDCS*, 2019.

[52] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *DSN*, 2018.

[53] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. Popularity prediction of Facebook videos for higher quality streaming. In *ATC*, 2017.

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

[55] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[56] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scoot Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale Web services. In *OSDI*, 2016.

[57] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.