Achieving Convergent Causal Consistency and High Availability for Cloud Storage

Yu Tang, Hailong Sun, Xu Wang, Xudong Liu

School of Computer Science and Engineering, Beihang University, Beijing, China 100191

Abstract

The tradeoff between consistency and availability is inevitable when designing distributed data stores, and today’s cloud services often choose high availability instead of strong consistency, leading to visible inconsistencies for clients. Convergent causal consistency is one of the strongest consistency model that still remains available during system partitions, and it can also satisfy human perception of causality between events. In this paper, we present CoCaCo, a distributed key-value store that provides convergent causal consistency with asynchronous replication, since it is able to provide cloud services’ desired properties including high performance and availability. Moreover, CoCaCo can efficiently guarantee causal consistency by performing dependency checking only during handling read operations. We implement CoCaCo based on Cassandra and our experimental results indicate that CoCaCo provides performance comparable to eventually consistent Cassandra.

Keywords: Causal Consistency, Availability, Key-value Data Store

1. Introduction

Nowadays distributed data stores have become a fundamental infrastructure for large-scale cloud systems, and they usually replicate data partitions to achieve high scalability and availability[1]. Although strong consistency, high performance and availability are desired properties for many modern
data stores, the CAP theorem [2, 3] indicates that it is impossible to achieve both strong consistency and high availability simultaneously when network partitions occur, and PACELC [4] further explains the tradeoff between consistency and latency when network partitions are absent.

Although many existing cloud services have to embrace eventual consistency [5] for better performance and availability [6, 7, 8], stronger consistency is always desirable for system correctness. Therefore, many studies [9, 10, 11, 12, 13, 14] pay more attention to convergent causal consistency, which is proved to be one of the strongest consistency models that can be achieved together with high availability [15, 16]. Though weaker than linearizability [17], convergent causal consistency couples the virtues of causal consistency [18] and eventual consistency [5]. As a result, convergent causal consistency not only guarantees the write causality, but also ensures that all replicas converge to the same state, which is critical for implementing reasonable application behaviors.

Providing convergent causal consistency both within and across datacenters is meaningful. As many studies [19, 20, 21] indicate that network partitions are often inevitable both within and across modern datacenters, and the network redundancy approach still cannot mask all possible network partitions [20], designs that employ strong consistency models (e.g. linearizability [17]) can incur the unavailability problem when network partitions occur according to the CAP theorem [3]. Besides, systems employing strong consistency models may produce higher latency when network partitions are absent [4].

In this work, we present CoCaCo, a distributed key-value store, which provides convergent causal consistency and high availability guarantee both within and across datacenters based on asynchronous replication. We use asynchronous replication because it can return a success response immediately after performing an update at one replica, without waiting for acknowledgements of replication messages from all replicas. Thus our work can provide both high availability and low latency. However, to achieve causal consistency with asynchronous replication, there are two main challenges: (1) how to efficiently deal with stale reads that violate causality; (2) how to efficiently guarantee causality during write operations.

Firstly, a modern datacenter often deploys multiple replicas locally to be fault-tolerant, thus the states of replicas in a datacenter may diverge as of the continuous coming of update requests due to asynchronous replication, and a client may observe a stale replica state that violates causal consistency within
the local datacenter. To address this issue, we employ causal dependency checking to detect stale reads that violate causal consistency, and if they exist, the coordinator will enforce a second-round read involving all replicas to eliminate them.

Secondly, how to guarantee write causality efficiently is a considerable problem [9, 13], and causal dependency checking is a common method to solve the problem [9, 10, 12]. In this work, a dependency checking strategy which is suitable for asynchronous replication is proposed: we choose to delay dependency checking, which is often performed when handling writes in the previous design, until handling reads. Therefore, the system can leverage this delay to propagate the updating states, which benefits the overall performance. Furthermore, CoCaCo only needs to perform dependency checking for the reading item instead of its dependent items, which avoids sending dependency checking messages to relevant data partitions [9, 10, 12], and thus additionally benefits system performance.

Therefore, CoCaCo is designed to (1) employ asynchronous replication both within and across datacenters for high availability and performance, (2) efficiently guarantee causal consistency by performing dependency checking only during handling read operations. Finally, we implement the CoCaCo prototype based on Apache Cassandra [7], a production-ready data store, and make experiments within and across datacenters. Experimental results indicate that in terms of performance, CoCaCo is comparable with eventual-consistent-configured Cassandra, and outperforms RYW-consistent-configured Cassandra, especially for read-most workloads.

In this paper, we make the following contributions:

- We present the design of CoCaCo to obtain convergent causal consistency both within and across datacenters based on asynchronous replication, and obtain high availability and performance for cloud storage.

- We demonstrate how CoCaCo handles writes and reads efficiently to guarantee causal consistency. CoCaCo delays dependency checking until handling reads and never blocks writes. Moreover, we prove that causal consistency is obtained in CoCaCo, and show how to support read-only transactions in CoCaCo.

- We implement the CoCaCo prototype based on Apache Cassandra. Our experimental results show that CoCaCo introduces negligible overhead compared with eventual-consistent-configured Cassandra, and achieves
47.8% ~ 167.5% higher throughput than RYW-consistent-configured Cassandra for read-most workloads, in both LAN and WAN environment.

Our earlier work entitled “Achieving Convergent Causal Consistency and High Availability with Asynchronous Replication” in the poster session of IWQoS 2016 [22] presents an approach to achieve convergent causal consistency. We have extended the paper and added significant materials. Firstly, we have introduced how to support read-only transactions in CoCaCo. Secondly, we have given a much more detailed presentation about the system design and implementation, and have extended our discussion, including an informal proof of the correctness of CoCaCo, how to handle failures and network partitions, and the durability of CoCaCo. Furthermore, we have evaluated CoCaCo with more extensive experiments, including measuring response latency, calculating the rate of second-round reads and dependency number of each item for various workloads, and evaluating the performance of CoCaCo in a WAN environment.

The rest of the paper is organized as follows. Section 2 presents the related work. In Section 3, we describe the design of CoCaCo. Section 4 discusses the correctness of CoCaCo and how to handle failures in CoCaCo. In Section 5 we provide the experimental evaluation. Section 6 summarizes our work and points out future works.

2. Related Work

The complex tradeoffs between consistency and availability [2, 3] or consistency and performance [4, 23] play an important role in the emergence of a large number of distributed data stores and consistency models. Just as CAP [2] and PACELC [4] indicate, there is no consistency model that can satisfy all possible scenarios. Linearizability [17] is probably the most intuitive model for application developers, but it is incompatible with availability during network partitions [3]. Moreover, even though when network partitions absent, linearizability is expensive to support in a replicated system as it needs to totally order all writes and thus restricts concurrency and leads to terrible performance. Systems [24, 25] employing other strong consistency models also have to sacrifice availability or performance as those that choose linearizability. That is often too expensive for geo-replicated systems. Therefore, data stores in wide area environment usually choose to embrace
availability and performance instead of strong consistency. In this paper, we primarily focus on convergent causally consistent stores as they can remain available during network partitions [15].

Causal consistency derives from the concept of “happens-before” from Lamport’s work [26]. Many system designers have recognized the value of causal consistency, and causally consistent systems including Lazy Replication [27], the ISIS toolkit [28], Causal Memory [18], Bayou [29], PRACTI [30] arise successively. However, those systems assume single-machine replicas and do not support data partitions, and therefore, they cannot scale to manage large-scale data.

Recent works have rekindled their interest in causally consistent systems. COPS [9] presents a solution on achieving causal consistency in a geodistributed data store. Clients in COPS store every accessed item’s version and dependencies as dependency metadata and attach this metadata with each write operation sent to the local datacenter. Then servers check dependencies by sending check message to other dependent partitions within each datacenter when processing write operations, and writes must wait until all dependencies are satisfied in the local datacenter before committed. Eiger [10], a follow-up work on COPS, supports a richer data model and powerful transaction, but retains the dependency checking strategy. Orbe [12] uses two-dimensional dependency matrices to track dependencies, using less dependency metadata. However, Orbe also adopts the similar dependency check strategy as COPS and Eiger. GentleRain [13] tracks causal consistency by leveraging loosely synchronized physical clocks and reduces dependency metadata. Moreover, GentleRain eliminates dependency checking messages and achieves better throughput compared to previous works. However, in the case of failures on any server, query will only observe data items that are stored before the failure happens. Cure [14] provides stronger semantics, including general transactions and support for confluent data types, compared to previous works [9, 10, 12, 13]. Orbe, GentleRain and Cure assume messages are propagated in order. Moreover, all these works [9, 10, 12, 13, 14] assume strong consistency within each datacenter, which will result in concerns for availability or latency during network partitions or jitter according to CAP [2] and PACELC [4]. Studies [19, 21] show that network partitions can occur within and across modern datacenters, and network redundancy is “not entirely effective” to mask network partitions [20]. Therefore, adopting causal consistency, instead of strong consistency, within and across datacenters is meaningful, especially to cloud services that choose availability.
However, the approaches proposed by those works [9, 10, 12, 13, 14] cannot be easily applied to guarantee causal consistency within each datacenter. Those approaches have the same or logically equal assumptions including (1) a client (web servers) only accesses servers in the same data center, and (2) linearizability is guaranteed within each datacenter. The two assumptions can guarantee that a client will update and access the same logical replicas. While it breaks the second assumption to apply their methods of achieving causal consistency across datacenters to a single datacenter. In that situation, a client may observe stale replicas that violate causal consistency, and those approaches do not address it.

ChainReaction [31] implements causal consistency within each datacenters by leveraging a variant of Chain Replication [32]. However, the dependency checking strategy of ChainReaction is the same with COPS and Eiger, and thus incurs considerable performance degradation. The Bolt-on approach [11] also provides causal consistency within and across datacenters. It inserts a shim-layer between the data store and the application layer and separates architectural concerns of liveness from safety [33]. To provide liveness property, the Bolt-on approach does not modify the underlying eventually consistent data store, which may return stale items that violate causal consistency. Therefore, to guarantee causal consistency, the shim-layer, which is located in client machines, must store all observed writes including their causal dependencies and item values to provide “stickiness” [5], and there is no safe strategy to remove those writes unless they are no longer required [11], which makes each client machine store large number of data items as the system runs. SwiftCloud [34] provides causal consistency via client-side cache backed by the cloud and adopts a design that can reduce causal metadata size. The main limitation of SwiftCloud is that it is designed for fully-replicated datacenters, and thus does not support data partitions, which means the scalability in server-side is restricted. PCSI [35] gives a solution on achieving the snapshot isolation [36] based transaction support with causal consistency in partially replicated databases. However, the snapshot isolation is incompatible with availability [19]. Moreover, PCSI uses a designated conflict resolver site to check update conflicts for each data item and prohibit update conflicts, which leads to more coordination messages and restricts the performance. In pursuit of the snapshot isolation, desired properties including performance and availability are sacrificed in PCSI, and that is different from CoCaCo.
3. Design

In this section, we present the design of CoCaCo. Firstly, we give a global view of CoCaCo, and then introduce the client layer and the storage layer respectively. After that, we demonstrate how to handle write and read operations within a datacenter. Then we show how to provide read-only transactional semantics in CoCaCo. Finally, we indicate that the previously described design can also be extended to multiple datacenters environment.

3.1. Design Overview

CoCaCo is a key-value data storage system designed to run on one or more datacenters, and within each datacenter we assume a full copy of data items is stored. In each datacenter, consistent hashing is used to distribute and locate data items and one or more replicas are deployed, as shown in Figure 1. To pursue high performance and availability, asynchronous replication is employed both within and across datacenters.

CoCaCo consists of the client layer and the storage layer. The client layer is composed of web servers. While the storage layer contains many data nodes that reside in one or more datacenters. A web server needs to use a client library to write or read data items into or from the storage layer, and a client should communicate with data nodes in the same datacenter first. However, a client can send requests to other datacenters when the underlying storage layer within the same datacenter becomes unavailable. A client must wait for the corresponding response of the current request before sending a next
request. To track and check causality, client libraries and data nodes need to store and exchange extra metadata compared with eventually consistent systems. To reduce resource and performance overhead, an optimization strategy is applied.

3.2. The Client Layer

The client layer consists of web servers, and a web server contains the client/application program and a CoCaCo client library. The client program is responsible for running code on behalf of end users (e.g., browsers) and meanwhile capturing causal relationship between operations. Each CoCaCo client library maintains in-memory dependency metadata, and is used by the corresponding web server to communicate with underlying data nodes.

We denote the causal order relationship between two operations by “⇝”. For any operation $a$ and $b$, $a ⇝ b$ means $a$ is a dependency of $b$. Causal relationships are captured in two ways: via potential or explicit causality [37].

For potential causality [18, 26] between any two operations $a$ and $b$, $a ⇝ b$ if one of the following rules holds: (i) order within each thread, e.g., if $a$ and $b$ are handled in the same thread, $a$ must happen before $b$, (ii) reads-from, e.g., if $a$ is a write operation and $b$ is a read operation, $b$ must read the value written by $a$, (iii) transitivity, e.g., there is a operation $c$ that $a ⇝ c$ and $c ⇝ b$. To capture potential causality, all data items a client writes and reads should be considered as dependencies of the following operations.

Tracking explicit causality offers a more flexible approach. Under explicit causality, user interfaces can be provided to application developers to define causal relationships between operations [11, 27]. Therefore, the number of dependencies in tracking explicit causality are often much less than that in tracking potential causality, and thus results in better performance and less metadata [37]. Considering its benefits, we choose to track explicit causality in this work. Explicit causality is often captured through user interfaces. For example, user A can click the corresponding “reply” button after one of user B’s comments to reply B with a new comment. Thus the client program is aware of the causal relationship between the two comments. Otherwise, the client program will ignore causal relationships between comments unless it captures potential causality. The example also indicates a limitation of employing explicit causality: application programmers must consider how to merge a capturing strategy into their application logic to capture explicit dependencies [37]. For example, the programmers need to define the causal
relationship of a comment and its replied comment besides implementing the basic functionality of comment replying. Therefore, programmers need more efforts to design and implement the applications, although, in many cases, explicit causality can be easily captured [37].

A client library maintains in-memory dependency metadata. The metadata consists of a set of \(<key, fts, deps>\) tuples, and each tuple contains the corresponding causal information of the item whose key is \(key\). \(fts\) represents the version of the item, while \(deps\) consists of \(<key, fts>\) tuples and each of them represents a dependent write operation. There are two core interfaces in the client library:

- boolean \(\leftarrow\) put_after\((key, value, depKeys)\)
- \(value \leftarrow\) get\((key)\)

Compared with the \textit{put} interface in a typical key-value store, \textit{put\_after} takes an optional \textit{depKeys} argument, which is a set of item keys that the inserting item causally depends on.

Each client library maintains a Lamport timestamp [26]. After \textit{put\_after} is invoked, the client library combines the local Lamport timestamp with a unique number to form an artificial timestamp, denoted as \(ts\). The unique number can be the unique id or the random magic number the client library holds. Specifically, the Lamport timestamp takes up the high-order bits of \(ts\), while the unique number takes up the low-order bits. The low-order bits are used to make the \(ts\) global unique for different clients. Then the client library retrieves corresponding dependencies from local metadata and invokes the \textit{c\_put} operation. While after the \textit{get} interface is invoked by a client, the client library will get corresponding \(fts\) from local metadata and then invoke the \textit{c\_get} operation. Both \textit{c\_put} and \textit{c\_get} will be described in Section 3.3.

A client library needs to update local Lamport timestamp and metadata. The Lamport timestamp will plus one before processing a write operation, and it should be updated to the returned item’s timestamp if the latter is larger when handling a read operation. To describe how to update local metadata, we define the \textbf{merge strategy} for merging dependencies \(deps_2\) to dependencies \(deps_1\): for any \(<key_2, fts_2>\) \(\in\) \(deps_2\),

- If \(\nexists <key_1, fts_1> \in deps_1\) such that \(key_1\) equals \(key_2\), then add the tuple \(<key_2, fts_2>\) to \(deps_1\), or
• Else if $\exists <key_1, fts_1> \in deps_1$ such that $key_1$ equals $key_2$ and $fts_1 < fts_2$, then assign $fts_2$ to $fts_1$.

After getting (or putting) an item from (or to) the data store, the library will merge the corresponding information of the execution result to the metadata. The goal of the merging is to guarantee causal consistency and make the client see a progressing view as well. The client library does not check whether dependencies are satisfied, instead, it acts like an interface proxy from the outside. Algorithm 1 demonstrates the behavior of the interfaces in a client, and the details of the algorithm are described in Section 3.4.

3.3. The Storage Layer

The storage layer is built upon Apache Cassandra [7], and is composed of multiple data nodes. CoCaCo adopts the default partition strategy of Cassandra and divides the keyspace into multiple partitions according to consistent hashing. Each CoCaCo datacenter maintains a full copy of data items, and each data partition can be replicated to every datacenter for fault tolerance. Nonetheless, the correctness of CoCaCo does not depend on the data partition or data placement strategy as discussed in section 4.1, which means the CoCaCo approach is suitable for other data partition or data placement techniques.

All data nodes in CoCaCo are considered equivalent. When a node receives a request from a client, the node becomes a coordinator, and it is responsible for distributing requests to relevant replica nodes, collecting and handling the feedbacks from other nodes, and responding to the client. The CoCaCo data store exposes two main interfaces to the client library:

• $fts \leftarrow c\_put(key, value, ts, deps)$

• $<value, deps, fts> \leftarrow c\_get(key, fts)$

The interface $c\_put$ takes a $deps$ argument, which is the dependencies of the item $key$ and $deps$ contains a set of $<key, fts>$ tuples; $ts$ is the timestamp generated by the client. The coordinator chooses the larger one (denoted as $lts$) from the high-order bits of $ts$ (Lamport timestamp generated by the client) and local Lamport timestamp. If the local Lamport timestamp is smaller than $lts$, then the coordinator updates local Lamport timestamp to $lts$ to make it progress. Afterwards, The coordinator generates $fts$ by combines $lts$ with the low-order bits of $ts$ (the unique number generated
by the client): \( lts \) takes up the high-order bits of \( fts \), while the unique number takes up the low-order bits. By this means, CoCaCo orders the writes arriving at the same coordinator.

The function \( c.get \) needs an \( fts \) argument, which specifies the basic dependency (version) that the accessing item should satisfy. The data store will query relevant replica nodes to try to satisfy the condition, and then return value and corresponding causal dependencies \( deps \). However, if the condition cannot be satisfied, \( null \) will be returned. Every data node may store multiple versions of items. However, only the value corresponding to the recent version will return for a \( c.get \) operation, in order to provide a progressing view and guarantee convergence. Algorithm 2 presents the primary behavior of the interfaces in a data node, and the details of the algorithm are described in Section 3.4.

3.4. Handling Write and Read Operations

3.4.1. Write Process

A write process begins if a client library receives a \( put.after \) request. The client library needs to initialize arguments for the \( c.put \) operation before invoking the operation. So the client library retrieves corresponding \( fts \) for each item that is contained in the argument \( depKeys \) from local metadata, wraps those parameters to corresponding dependencies and adds those dependencies to the argument \( deps \) (whose initial value is empty) of \( c.put \). Moreover, dependencies of items that are contained in \( depKeys \) should also be merged to \( deps \) to maintain transitive causal relationships (to solve the problem of “overwritten histories” [11]). The argument \( ts \) of \( c.put \) is also generated by the client library. Afterwards, the client library invokes \( c.put \) operation.

When a coordinator receives a \( c.put \) request, it generates \( fts \) according to Section 3.3, and launches a replication stage. In a replication stage, the coordinator firstly distributes replicate messages, which piggyback a corresponding \( fts \), to relevant replica nodes. After a replica node receives a replicate message, it stores the value, \( fts \) and \( deps \) locally and then sends an acknowledgement feedback to the coordinator. After receiving an acknowledgement, the coordinator returns the corresponding \( fts \) to the client, and the client library will update its local timestamp and metadata as described in Section 3.2. Then the write process is over.
Algorithm 1 Operations at a client

metadata contains a set of triples <key, fts, deps>. Each triple is identified by key, and the corresponding fts or deps can be retrieved by invoking get_fts or get_deps function respectively.

1: procedure Put_After(key k, value v, depKes dks)
2:     deps ← Empty
3:     // Lamport timestamp takes up the high-order bits of ts,
4:     // while client id takes up the low-order bits.
5:     ts ← (LamportTimestamp.increaseAndGet(), ClientID)
6:     for each dk ∈ dks do
7:         merge(deps, <dk, metadata.get_fts(dk, default=0)>)
8:         merge(deps, metadata.get_deps(dk, default=Empty))
9:     send c_put ⟨k, v, ts, deps⟩ to a data node
10:    receive response ⟨fts⟩
11:    merge(metadata.get_deps(k, default=Empty), ⟨k, fts⟩)
12:    LamportTimestamp.setIfLarger(fts.getLamportTimestamp())

13: procedure Get(key k)
14:     fts ← metadata.get_fts(k, default=0)
15:     send c_get ⟨k, fts⟩ to a data node
16:    receive response ⟨value v, deps, fts⟩
17:    merge(metadata.get_deps(k, default=Empty), ⟨k, fts⟩)
18:    LamportTimestamp.setIfLarger(fts.getLamportTimestamp())

19: procedure MERGE(dependencies deps₁, dependencies deps₂)
20:     for each ⟨k, fts⟩ ∈ deps₂ do
21:         if deps₁.get(k)==null then
22:             deps₁.put(k, fts)
23:         else if fts > deps₁.get(k) then
24:             deps₁.put(k, fts)
Algorithm 2 Operations at a data node

// at coordinators
1: \textbf{upon receiving} c.put request \langle key k, value v, ts, deps \rangle
2: \hspace{1em} localTS \leftarrow \text{LamportTimestamp.increaseAndGet()}
3: \hspace{1em} \text{fts} \leftarrow \max(\text{localTS}, \text{ts.getLamportTimestamp()})
4: \hspace{1em} \text{LamportTimestamp.setIfLarger(fts)}
5: \hspace{1em} \text{send replicate} \langle key k, value v, \text{fts}, \text{deps} \rangle \text{ to corresponding replicas}
6: \hspace{1em} \text{wait until one replicate.ack from any replica}
7: \hspace{1em} \text{send} \langle \text{fts} \rangle \text{ to the client}
8: \textbf{end proceeding}

// at coordinators
9: \textbf{upon receiving} c.get request \langle key k, \text{fts} \rangle
10: \hspace{1em} \text{send fetch} \langle key k, \text{fts} \rangle \text{ to a replica}
11: \hspace{1em} \text{receive get_reply} \langle \text{value}, \text{deps}, \text{fts}' \rangle
12: \hspace{1em} \text{if} fts' \geq \text{fts} \text{ then}
13: \hspace{2em} \text{send response} \langle \text{value}, \text{deps}, \text{fts}' \rangle \text{ to client}
14: \hspace{1em} \text{else}
15: \hspace{2em} \text{send fetch} \langle key k, \text{fts} \rangle \text{ to all living replicas}
16: \hspace{2em} \hspace{1em} \text{wait until any response} \langle \text{value}', \text{deps}', \text{fts}'' \rangle \text{ where} fts'' \geq \text{fts} \text{ or}
17: \hspace{2em} \hspace{2em} \text{receiving responses from all living replicas}
18: \hspace{2em} \hspace{1em} \text{send response} \langle \text{value}', \text{deps}', \text{fts}'' \rangle \text{ to the client}
19: \text{end proceeding}

// at replicas
20: \textbf{upon receiving} replicate request \langle key k, value v, \text{fts}, \text{deps} \rangle
21: \hspace{1em} \text{persist} \langle k, v, \text{fts}, \text{deps} \rangle \text{ locally}
22: \hspace{1em} \text{send replicate.ack to the coordinator}
23: \textbf{end proceeding}

// at replicas
24: \textbf{upon receiving} fetch request \langle key k, \text{fts} \rangle
25: \hspace{1em} \text{obtain the latest version} \langle v, \text{deps}, \text{fts}' \rangle \text{ of key k locally}
26: \hspace{1em} \text{if} fts' \geq \text{fts} \text{ then}
27: \hspace{2em} \text{send response} \langle v, \text{deps}, \text{fts}' \rangle \text{ to the coordinator}
28: \hspace{1em} \text{else}
29: \hspace{2em} \text{send response} \langle \text{null}, \text{null}, \text{fts}' \rangle \text{ to the coordinator}
30: \textbf{end proceeding}
3.4.2. Read Process

A read process begins if a client invokes *get* operation. The client library retrieves *fts* corresponding to the argument *key* of *get* operation from metadata, to constitute the arguments of *c.get* operation. If the metadata does not contain the information of *key*, the *fts* is set to 0. After that, the client library invokes *c.get* operation.

After receiving a *c.get* request, the coordinator forwards the request message to a random (or the fastest) corresponding replica node. Then the replica node will retrieve the item locally and check the dependency: if the timestamp of the retrieved item is equal or larger than the *fts* carried in the request message, then the checking is passed; otherwise, the dependency checking is failed. If the dependency checking is passed, the replica node will send the value and dependencies to the coordinator; afterwards, the coordinator will send the result to the client. However, if the dependency checking is failed, the replica node will inform the coordinator, and then the coordinator raises a second-round read: the coordinator distributes the request to all the corresponding replica nodes and waits until it receives any response that indicates the dependency checking is passed; then, it sends the read result to the client library. After receiving the result, the client library will update its local timestamp and metadata as described in Section 3.2. Then the read process is over. CoCaCo performs dependency checking only for the reading item during handling read operations, which avoids sending dependency checking messages to other dependent data partitions. Furthermore, CoCaCo ensures that clients can also observe dependent items without violating causal consistency by performing dependency checking for the dependent items when reading them.

3.4.3. Convergence

The convergence property ensures that all replicas will converge to the same state eventually, but causal consistency alone does not guarantee the convergence property. Causal consistency does not intend to define a global order for operations. Therefore, inevitably, there are some causally unrelated operations, which are considered to be *concurrent*. Formally, if \( a \not\Rightarrow b \) and \( b \not\Rightarrow a \), then \( a \) and \( b \) are concurrent. Concurrent writes corresponding to the same item often lead to inconsistent data states and those writes are in conflict. Conflicts do not violate causal consistency, but can lead to undesired results. For example, a client might observe different values of an item time after time, even though no following writes corresponding to the item
comes any more. In this work, we use the “last-writer-wins” rule [38], widely adopted in other systems [9, 10, 31], to handle conflicts. When combined with causal consistency, to determine which writer is “last” is considerable: if $a_1$ and $a_2$ are write operations corresponding to the same item and $a_1 \rightarrow a_2$, then $a_2$ is “last” and the state should not converge to $a_1$; while an eventually consistent data store may make the state converge to $a_1$ and thus violates causal consistency. Nonetheless, the convergence property guarantees that all corresponding replicas will eventually converge to $a_2$, and thus a client may still witness $a_1$ before the replicas converge to $a_2$ [11]. To provide the convergence property guarantee, CoCaCo also faces the peak throughput problems of “all-to-all replication” [37].

As an implementation of “last-write-wins”, timestamp based reconciliation [6] is performed by many eventual-consistent data stores [6, 7] to obtain convergence property, because it is intuitive and instrumental: the write with the largest timestamp is chosen as the winner of conflicting ones. In CoCaCo, each data node also employs timestamp based reconciliation, and timestamps of causally related write operations are guaranteed to be incremental. Thus states in CoCaCo are convergent without violating causal consistency.

3.5. Read-Only Transactions

Read-only transactions with causally consistent guarantee give clients a consistent view of many items that may reside in multiple servers. Without read-only transactions, clients may observe inconsistent items in some situations. For example, user A updates the permission of an album from “public” to “private”, and then uploads some photos to the album. Afterwards, another user B reads the album via two separate reads: the first read observes the stale state of the permission, whose value is “public”, while the second read returns the added photos. Thus, B can observe the photos whose owner thinks they are invisible to other users. By providing a consistent view, read-only transactions with causally consistent guarantee in CoCaCo can prevent the undesirable outcome. CoCaCo introduces additional operations in the client library and the data store to provide read-only transactions:

- In client libraries: \{<key, value>\} $\leftarrow$ get\_tx\{\{key\}\}.
- In data nodes: \{<key, value, deps, fts>\} $\leftarrow$ c\_get\_tx\{\{key, fts\}\}.

A CoCaCo client library needs to retrieve the corresponding $fts$ from local metadata for each item when get\_tx is invoked. If the metadata does
not contain the \( ft \)s for an item, the \( ft \) is set to 0. After the \( ft \)s of all the items are set, the client library invokes \( c.get.tx \).

Then, a data node (the coordinator) receives the \( c.get.tx \) request, generates corresponding read requests for every item and forwards the requests to the corresponding replicas. For each item that is contained in the transactions, the read process is similar to the basic read process. After obtaining all the items, the coordinator merges the dependencies of those items according to the merge strategy described in Section 3.2. Denote the merge result as \( m.r \). The obtained items are \textit{consistent} if for any item, denoted as \( i.tm \), in the transaction, either of the following conditions can be satisfied:

1. \( \exists <i.tm.key, ft.s_1> \in m.r \text{ such that } ft.s_1 \leq i.tm.ft \), or
2. \( \nexists <i.tm.key, ft.s_1> \in m.r \).

Otherwise, that means there are some stale returned items which lead to an inconsistent view. If all the returned items are consistent, then the coordinator can return those items to the client. Otherwise, the coordinator must raise a second-round read. In the second round, the coordinator has already known the required \( ft \)s of the items that are stale, and starts a read process similar to the basic read process for each stale item. A replica should return the item whose \( ft \) is equal to the required timestamp during the second round. After the coordinator receives all the items, it can respond the client.

3.6. Optimization

Compared with eventually consistent data stores, CoCaCo transmits and stores extra data, e.g. dependencies, and can influence the entire performance. We employ a general idea to reduce the dependency metadata size without violating causal consistency. The key point of the idea is that, once an item (without loss of generality, denoted as \( x \)) is replicated to all replicas, a writing item depending on \( x \) does not need to merge the tuple \( <x.key, x.fts> \) to \( deps \) as described in section 3.3, and there is also no need to check \( x.ft \)s during reading \( x \), unless there are some new updates on \( x \). Additionally, client libraries can remove the dependencies that represent the item \( x \) from local metadata. Therefore, perceived unnecessary dependencies will not be transmitted and persisted and the system performance can be optimized. The idea is also employed in COPS [9] and Orbe [12].
3.7. Multiple Datacenters Environment

Servers in a single datacenter are equivalent in CoCaCo. Furthermore, the correctness of CoCaCo does not depend on the data partition or data placement strategy. Therefore, achieving convergent causal consistency in a multiple datacenters environment is straightforward: for each datacenter, just regard nodes in remote datacenters as nodes in the same datacenter but spending more communication time. Then the design described ahead can be also applied in a multiple-datacenter environment. Furthermore, we can add restrictions in CoCaCo to obtain some better properties. For example, a client can communicate with the local datacenter first to reduce latency and improve user experience.

Compared with the network latency between nodes within a single datacenter environment, the latency between datacenters is much larger [19], and seems to affect the performance of CoCaCo since a second-round read may take more time to get a required result. However, according to our experimental results (Section 5.6), CoCaCo still introduces small overheads compared to an eventually consistent configuration of Cassandra, because the probability of second-round read is often low. Additionally, the network latency between datacenters does not remarkably influence the write performance since CoCaCo employs an asynchronous replication strategy.

4. Discussion

4.1. Correctness

In this subsection, we firstly prove that CoCaCo without the optimization strategy can guarantee causal consistency. Afterwards, we demonstrate that applying the optimization strategy does not violate causal consistency. Failures and network partitions are not considered in the process of proving, but they will be discussed in 4.2. The proof mainly focuses on the basic write and read operations.

For the item $x$ that is observed in a client, we denote its corresponding write operation as $wop(x)$. We also assume “⇝” is irreflexive or $a \nleftrightarrow a$ for any operation $a$ as [26].

We define a “forcibly-happens-before” relationship between write operations in CoCaCo and denote it as “⇒”. For write operations $a$ and $b$, we define $a \Rightarrow b$ if and only if $a.fts < b.fts$. If $a \sim b$, then $a \Rightarrow b$, but the converse does not hold.

Lemma 1: $\forall a, b, c$, if $a \nleftrightarrow b$ and $b \nleftrightarrow c$, then $a \nleftrightarrow c$. 

17
When \( a \not\rightarrow b \) and \( b \not\rightarrow c \), \( a.fts \geq b.fts \geq c.fts \) holds, and thus \( a \not\rightarrow c \) holds.

Without considering the optimization strategy, according to the merge strategy described in 3.2, we can prove the correctness of Lemma 2.

**Lemma 2:** \( \forall a \) and \( b \), if \( a \bowtie b \), then \( \exists \text{dep} \in b.\text{deps} \) such that \( \text{dep.key} \) equals \( a.\text{key} \) and \( \text{wop} (\text{dep}) \not\rightarrow a \).

Without the optimization strategy, \( \exists \text{dep} \in b.\text{deps} \) such that \( a.\text{key} \) equals \( \text{dep.key} \) according to the merge strategy. By reduction ad absurdum, if \( \text{wop} (\text{dep}) \rightarrow a \), then \( \text{wop} (\text{dep}).fts < a.fts \), but that violates the merge strategy. Thus Lemma 2 is proven.

**Causal consistency is guaranteed in CoCaCo:** given a write operation \( a \), \( \forall b \) where \( b \bowtie a \), if a client observes \( a \), then the client should be able to observe an item denoted as \( x \) where \( x.\text{key} \) equals \( b.\text{key} \) and \( \text{wop}(x) \not\rightarrow b \).

If a client observes \( a \) and then wants to get the item whose key is \( b.\text{key} \) where \( b \bowtie a \), according to Lemma 2, \( \exists \text{dep} \in a.\text{deps} \) such that \( b.\text{key} \) equals \( \text{dep.key} \) and \( \text{wop} (\text{dep}) \not\rightarrow b \). The client library will invoke \( \text{c.get} (\text{dep.key}, \text{dep.fts}) \).

According to the read process, the CoCaCo store will return an item \( x \), where \( x.\text{key} \) equals \( \text{dep.key} \) and \( \text{wop}(x) \not\rightarrow \text{wop}(\text{dep}) \). As \( \text{wop}(x) \not\rightarrow \text{wop}(\text{dep}) \) and \( \text{wop}(\text{dep}) \not\rightarrow b \), \( \text{wop}(x) \not\rightarrow b \) holds according to Lemma 1.

The CoCaCo system handles writes and reads as described in Section 3. For any two write operations \( a \) and \( b \), the CoCaCo system ensures that \( a.fts < b.fts \) if \( a \bowtie b \). In other words, CoCaCo guarantees that causally related operations have incremental timestamps, and this property is necessary when proving Lemma 1. The CoCaCo system guarantees the property in a simple way: when a client observes \( a \bowtie b \), the client will assign a timestamp \( ts \) that is larger than \( a.fts \) to \( b \) according to the timestamp updating and generating strategy in clients. Additionally, after a coordinator receives the request of \( b \), the coordinator will determine the \( b.fts \), which is larger than or equal to \( ts \). Therefore, \( a.fts < b.fts \) holds. The correctness of Lemma 2 depends on the merge strategy described in Section 3.2. Therefore, a CoCaCo client library, which stores all observed dependencies, updates local dependencies after getting (or putting) an item from (or to) a data node according to the merge strategy. Moreover, to guarantee causal consistency, CoCaCo also needs to ensure the version of each observed item is no less than the corresponding version determined by the client when handling a read operation. The CoCaCo system guarantees the property by implementing the design described in Section 3.3 and 3.4.2.

As the proof does not involve the data partition and data placement
strategies, the correctness of the CoCaCo approach does not depend on them.

Once a data item, without loss of generality, denoted as \( x \), is replicated to all replicas, following reads about \( x.key \) will not observe a version whose timestamp is smaller than \( x.fts \). Therefore, without violating causal consistency, write operations that depend on \( wop(x) \) do not need to merge the tuple \( <x.key, x.fts> \) to \( deps \).

4.2. Handling Failures and Network Partitions

We assume that during processing a basic read operation (or a read-only transaction), the coordinator can contact at least one replica that can pass dependency checking for the (or every) item it attempts to access. Otherwise, the coordinator will return \textcolor{red}{null} or throw an exception to the client.

Client Failures. CoCaCo’s clients are actually front-end web servers, which send write or read requests to data servers to operate data items. CoCaCo’s clients do not communicate with each other, and dependency metadata resides in clients are relatively independent. Therefore, a client’s failure does not influence other clients. If a client fails, we expect service providers to direct requests that should have been sent to the failed client to other living clients. However, the metadata reside in the failed client can be lost and this is unavoidable for systems (e.g. [9, 10, 11, 12, 13, 14]) that store dependency metadata in clients. That may make end users observe stale values. Nonetheless, modern Internet applications often employ caches, which store previous responses from servers, at terminals to improve user experience. Terminal programs can merge caches with incoming responses and pick the newer version to alleviate stale reads. Moreover, the causal relationships of handled operations are still hold, and the causal relationships of following operations will also be maintained.

Inner-Datacenter. In each datacenter, replicas are employed to provide fault tolerance. Each server can be the coordinator for any request from any client. Therefore, a client can choose any non-failing server as the coordinator. During writing in CoCaCo, the coordinator only needs to wait for one response from any replica; while during reading, the coordinator waits until observing an item that passes the checking. Therefore, CoCaCo can continue to process requests and respond clients unless the assumptions described earlier in this section are broken. However, server failures or network partitions can make the optimization strategy unworkable for items that reside in the failed or unreachable server(s), but they do not affect the achievement of convergent causal consistency. Moreover, after failures recovery, the
optimization strategy can continue to work.

**Inter-Datacenter.** Datacenter failures can raise by power outage, network partitions or even disasters. During transient failures, clients can wait until the datacenter is recovered or reconnect to other living datacenters; while during permanent failures, clients have to choose the latter choice. Though clients are reconnected to other datacenters, causal consistency is also guaranteed because client libraries maintain dependency metadata locally. After transient failures, datacenters can start a recovery stage to exchange and update data states, making replicas consistent. However, permanent failures may result in data loss. Only data that is not propagated to other datacenters but committed in the failed datacenter will be lost. This is inevitable for data stores (e.g. [9, 10, 11, 12, 13, 14]) that do not synchronously replicate data to other datacenters. To avoid data loss due to permanent datacenter failures, a system needs to guarantee globally strong consistency and replicate data to remote datacenters synchronously before responding clients [39, 40, 41], at a cost of decreased availability and increased latency [19].

### 4.3. Durability

In the previous description of CoCaCo, when handling a write operation, a coordinator only needs to wait for one acknowledgement from replicas, which may influence data durability even when a single server fails. In order to enhance durability, a coordinator can wait until more acknowledgements are received from replicas during handling write operations. To survive $f$ simultaneous failures, a coordinator needs to guarantee data is successfully persisted on $f + 1$ replicas, and therefore should wait for $f + 1$ acknowledgements from replicas during a replication stage. However, waiting for more acknowledgements affects the system performance and availability. The tradeoff between durability and performance/availability depends on the demands of the application.

### 5. Evaluation

#### 5.1. Implementation and Experimental Setup

Our CoCaCo prototype is implemented based on Apache Cassandra [7]. Figure 2 presents main modules in a data node and summarizes the relationship of each module with Cassandra. We directly reuse the data partition management module of Cassandra to distribute data across multiple data
nodes, the data replication management module to manage multiple replicas, and the storage module to persist and fetch items from the local disk. Additionally, we modify the client-server listener module of Cassandra to provide two interfaces described in Section 3.4.1 and 3.4.2, and the operation handler module to implement process control introduced in Section 3.4.1 and 3.4.2. Moreover, we modify the server-server communication module to support causality-related data exchange besides original data exchange. Furthermore, we add a timestamp management module to manage the local Lamport timestamp and determine the version of an inserting item, and add a dependency inspector module to check whether an observed item can satisfy the required version. The CoCaCo client library is implemented based on Cassandra client library: we add an in-memory dependency management module to manage all observed dependency, and implement two functions (put, after and get) as presented in Section 3.2.

We choose to implement CoCaCo based on Cassandra because it is a production-ready data store that adopts a symmetric and decentralized design, which can lead to a more scalable and more available system [6]. As in Figure 2, we directly reuse the management modules of data partition and replication, which contribute to the symmetric and decentralized implementation. However, as discussed in Section 4.1, the correctness of our approach does not depend on those two modules, and therefore, the design can also be applied to other distributed data stores with other data partition or replication strategies. Nonetheless, to provide high availability, data partition and replication strategies that conduct to symmetry and decentralization are suggested.

Cassandra mainly provides two consistency guarantees: eventual consistency and “read your writes” (or “RYW”) consistency. Eventual consistency is the default consistency model that Cassandra provides, and the eventual-consistent-configured Cassandra (denoted as Cassandra-Eventual) is commonly configured as $W=R=1$ [42], where $W$ and $R$ represent the number of responses that are needed to be received before coordinators respond clients for writes and reads respectively. Cassandra’s RYW consistency is weaker than causal consistency [11]. The RYW-configured Cassandra (denoted as Cassandra-RYW) must ensure that $W + R > N$, where $N$ is the total number of replicas.

In the experiments, each server node has one eight-core CPU of 2.0GHz and 16GB RAM. For the experiments in the local area network (LAN), CoCaCo and Cassandra are deployed on 5 servers, connected by an 1Gbps
Ethernet, and each data item has 3 replicas. While 6 servers are used to deploy CoCaCo and Cassandra for the experiments in the wide area network (WAN). The 6 servers are within the same cluster, but we partition 6 servers equally into 2 logical datacenters by adding network latency between two datacenters [12]. We evaluate the system with the Yahoo! Cloud Serving Benchmark [43]. By default, clients use 20-byte row keys. To expose the effects of dependency metadata overheads of CoCaCo, we set the size of data
value to 1 byte. Client threads in the same node share the dependency data to reduce memory usage. Before issuing a write, a client chooses an observed item randomly and regards it as the dependency of the write. The write also needs to attach the dependent item’s all dependencies as described in 3.4.1. We startup the optimization strategy to clear unnecessary dependencies. In our implementation, if a client library knows that an item \( x \) has already replicated to all the replicas, it will set \( x.fts \) to 0 in the local metadata, and following writes depend on \( x \) will omit \( x \) unless \( x.fts \) is updated to a non-zero value. At the beginning of each experiment, we clear the existing items and reinsert 10 million items into CoCaCo and Cassandra. We conduct the experiments on workloads with different read proportions and request distributions, including uniform and Zipfian distribution. By default, the value of the exponent (referred to as the factor) in Zipfian distribution is set to 0.88, which makes 80% of the requests relate to 20% of the 10 million items. The larger the factor is, the more concentrated requests are. Cassandra-Eventual is configured as \( W=R=1 \), while Cassandra-RYW is configured as \( W=R=\lfloor N/2 \rfloor +1 \), which can tolerate a failure of at least one replica when the number of replicas \( N \) is larger than two.

In the following experiments, we firstly present reads in Cassandra-Eventual may violate causal consistency, while reads in CoCaCo do not. Afterwards, we compare the performance of CoCaCo with that of Cassandra-Eventual and Cassandra-RYW in the LAN environment, and then in the WAN environment.

5.2. Correctness

Theoretically, reads in Cassandra-Eventual may violate causal consistency because Cassandra-Eventual only provides eventual consistency. We show stale reads that observe stale items that violate causal consistency may happen in Cassandra-Eventual, and meanwhile demonstrate there is no such reads in CoCaCo. In this experiment, we fix the number of client threads and then detect whether the observed items violate causal consistency. Table 1 presents the proportion of stale reads that violate causal consistency against all reads as read proportion varies under the Zipfian distribution for Cassandra-Eventual. As the results demonstrate, the percentage of stale reads that violate causal consistency decreases as the read proportion increases. Because the increasing of read proportion means a lower write proportion and items are updated less frequently, reads are less likely to observe stale replicas that the most recent writes do not synchronize in time.
Table 1: The percentage of stale reads that violate causal consistency.

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Read Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>Cassandra-Eventual</td>
<td>0.16%</td>
</tr>
</tbody>
</table>

We vary the Zipfian factor and find the stale proportion increases with the increasing of the factor, since updates are more concentrated on few items. While stale proportions are much less under the uniform distribution, as updates are uniformly distributed to all items. Stale reads that violate causal consistency do not appear in CoCaCo according to the experimental results under different workloads. Moreover, the number of client threads also influences the stale proportion. In our experiments, the stale proportion increases as the thread number becomes larger, because updates become more frequent accordingly.

Cassandra-RYW also cannot guarantee causal consistency [11], and we present a scenario (as in Figure 3) to show the configuration of W=R=2 and N=3 may violate causal consistency. However, if combined with read repair [44] and hinted handoff [45] in Cassandra, stale read that violates causal consistency does not appear in our experiments. After both read repair and hinted handoff in Cassandra-RYW are disabled, we can observe stale reads, though the probability is very low.

Figure 3: The configuration of W=R=2 and N=3 may violate causal consistency. The write requests that are sent to replica\textsubscript{2} and replica\textsubscript{3} are delayed, which makes coordinator\textsubscript{2} observes a stale item.
5.3. Read Proportion

We begin by evaluating the performance on workloads with different read proportions and request distributions. Fig 4 shows the corresponding throughput-workload curves.

As in the Fig 4(a), the curves demonstrate the peak throughput with different read proportions under the uniform distribution. Compared to the Cassandra-Eventual, the maximum overhead brought by CoCaCo is less than 4.7%. Compared to Cassandra-RYW, CoCaCo achieves much better performance; especially for read-most (read proportion ≥ 60%) workloads, the throughput of CoCaCo is about 63.0%−87.1% higher. The main reason why the performance of reads is lower than that of writes, is that Cassandra nodes append data in CommitLog and memory and then flush data into disk asynchronously, while reads may have to access disks [46]. When the read proportion is small, the gap between the curves of CoCaCo (or Cassandra-Eventual) and Cassandra-RYW is small, because the number of messages that a coordinator needs to send is the same (actually $N = 3$) for all three configurations during writing. The performance of CoCaCo (or Cassandra-Eventual) is better because it waits for less acknowledgements. While the gap between the curves of Cassandra-RYW and others is enlarged as the read proportion increases, because the number of messages that a coordinator needs to send is different for those configurations during reading: for Cassandra-Eventual and each first-round read in CoCaCo, a coordinator only needs to send one message to replicas, while the message number is two for Cassandra-RYW as a coordinator need to communicate with a majority of replicas. Though the message number is $N = 3$ for second-round reads in CoCaCo, the rate of second-round reads is small (see Table 2).

Similarly, in the Fig 4(b), the gap of throughput between CoCaCo and Cassandra-RYW becomes large as read proportion increases (CoCaCo is 47.8%−80.6% higher for read-most workloads), and CoCaCo achieves competitive performance compared to Cassandra-Eventual (about 1.6%−5.4% overhead). The reason why the peak throughput of reads is higher than that of writes in this workload is that much more operations hit the cache of data store, as operating items are more concentrated under the Zipfian distribution.

There are two observable factors that can influence the performance overheads of CoCaCo: the ratio of second-round reads (denoted as $rate_{sr}$) and the average actual number of dependencies of operated items (denoted as $depNumber_{aa}$). A dependency takes up 28 Bytes in the experiments (20

25
Bytes for `key` and 8 Bytes for `fts`). Intuitively, a client will observe larger read latency if a coordinator have to issue a second-round read. While if $depNumber_{aa}$ is larger, the data nodes will cost more time to transmit the message and operate (persist or access) data in disks.

We monitor the rate of second-round reads for different workloads. Table 2 presents the results: the rate under the Zipfian distribution is much higher than that under the uniform distribution, and the rate increases as the read proportion decreases under the same distribution. Under the Zipfian distribution, a client is more likely to read an item which was written not long ago, and therefore, the possibility that the first-round read cannot pass the dependency checking is higher, making second-round reads more frequent compared with that under the uniform distribution. Similarly, under the same distribution, more frequent writes mean the possibility of accessing an item that has not been replicated to all replicas is greater, and therefore, the rate is higher.

Moreover, we also calculate $depNumber_{aa}$ for each workload. Table 2 demonstrates the results: the $depNumber_{aa}$ under the Zipfian distribution is much larger than that under the uniform distribution; the $depNumber_{aa}$
Table 2: The \( r_{sr} \) and \( dep_{Number\text{aa}} \) for different read proportions.

<table>
<thead>
<tr>
<th>Property</th>
<th>Req Distrib</th>
<th>Read Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>( r_{sr} )</td>
<td>Uniform</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td></td>
<td>Zipfian</td>
<td>0.69%</td>
</tr>
<tr>
<td>( dep_{Number\text{aa}} )</td>
<td>Uniform</td>
<td>0.061</td>
</tr>
<tr>
<td></td>
<td>Zipfian</td>
<td>0.204</td>
</tr>
</tbody>
</table>

decreases as read proportions increase. The reason is that, under the Zipfian distribution or as write proportions increase, dependent items of an operating item are more likely to be the items that are not replicated to all replicas and whose \( fts \) are not removed in the metadata of client libraries yet (according to the section 3.6), and thus client libraries must append the corresponding information (e.g. \textit{key} and \textit{fts}) of those dependent items in the requests.

When clients choose keys to access by following the uniform distribution, the rate of second-round reads is tiny and the average actual dependency number is small, and therefore the curve of CoCaCo is close to that of Cassandra-Eventual as shown in Fig 4(a); while under the Zipfian distribution, both the rate of second-round reads and the dependency number become a little larger and thus the gap between the curves of CoCaCo and Cassandra-Eventual becomes a little larger, as shown in Fig 4(b). Nonetheless, the introduced performance overheads of CoCaCo is still small (less than 5.4%) compared with Cassandra-Eventual, and the peak throughput of CoCaCo is also much higher than that of Cassandra-RYW, especially for read-most workloads (47.8%~80.6% higher).

In the Fig 4, the gap between the curve of CoCaCo and Cassandra-Eventual is enlarged as the read proportion decreases. Except for the reason that \( r_{sr} \) and \( dep_{Number\text{aa}} \) are increased, another reason is that the garbage collection of memory in CoCaCo is more frequent than that in Cassandra-Eventual when handling writes, because CoCaCo needs more memory usage to store incoming dependencies and then release their memory space after dependencies are persisted. Fig 4 indicates that CoCaCo incurs little overhead.

Table 3 demonstrates the latency of three systems under the same workload whose read proportion is 80%. We choose the read proportion because reads are the most common operations in many real applications [40, 47]. To measure the latency fairly, we conduct the measurements under the similar
Table 3: Latency (in microseconds or $10^{-6}$ seconds) under the same workload (read proportion is 80%) with similar throughput.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>System</th>
<th>Operation</th>
<th>Average</th>
<th>50%</th>
<th>90%</th>
<th>95%</th>
<th>99%</th>
<th>99.9%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zipfian</td>
<td>CoCaCo</td>
<td>Read</td>
<td>616.4</td>
<td>595.6</td>
<td>880.7</td>
<td>921.0</td>
<td>1161.4</td>
<td>3365.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>409.0</td>
<td>312.1</td>
<td>563.4</td>
<td>594.2</td>
<td>795.2</td>
<td>3301.9</td>
</tr>
<tr>
<td></td>
<td>Cassandra-E</td>
<td>Read</td>
<td>608.7</td>
<td>584.0</td>
<td>879.7</td>
<td>917.7</td>
<td>1129.7</td>
<td>2283.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>380.9</td>
<td>288.3</td>
<td>543.3</td>
<td>569.7</td>
<td>751.3</td>
<td>1350.3</td>
</tr>
<tr>
<td></td>
<td>Cassandra-C</td>
<td>Read</td>
<td>816.8</td>
<td>823.8</td>
<td>1038.5</td>
<td>1213.5</td>
<td>1433.5</td>
<td>2328.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>580.6</td>
<td>554.5</td>
<td>648.0</td>
<td>784.8</td>
<td>972.5</td>
<td>1517.5</td>
</tr>
<tr>
<td>Uniform</td>
<td>CoCaCo</td>
<td>Read</td>
<td>752.4</td>
<td>652.7</td>
<td>969.0</td>
<td>1093.1</td>
<td>1419.0</td>
<td>3576.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>406.8</td>
<td>316.4</td>
<td>574.6</td>
<td>648.1</td>
<td>927.8</td>
<td>2883.8</td>
</tr>
<tr>
<td></td>
<td>Cassandra-E</td>
<td>Read</td>
<td>749.4</td>
<td>653.3</td>
<td>930.7</td>
<td>1003.6</td>
<td>1258.9</td>
<td>2032.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>394.9</td>
<td>299.9</td>
<td>557.6</td>
<td>597.9</td>
<td>810.9</td>
<td>1444.6</td>
</tr>
<tr>
<td></td>
<td>Cassandra-C</td>
<td>Read</td>
<td>1108.0</td>
<td>966.0</td>
<td>1391.4</td>
<td>1581.2</td>
<td>2071.6</td>
<td>3618.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>634.9</td>
<td>575.4</td>
<td>835.0</td>
<td>945.8</td>
<td>1192.6</td>
<td>1913.4</td>
</tr>
</tbody>
</table>

throughput for the three configurations. We conduct the experiments under the “similar” throughput instead of the “same” throughput, because we cannot make systems run at an exactly specified throughput. Table 3 indicates that CoCaCo incurs little overhead in terms of average latency, and the main reason is that both $rate_{sr}$ and $depNumber_{aa}$ are small when read proportion is 80%. However, the gap between CoCaCo and Cassandra-E is large when measured latency is 99.9%. One reason is that the dependency numbers of 0.1% operating items are large enough to greatly affect the performance of data transmission and accessing. The other reason is that second-round reads, though the probability is low, can disadvantageously influence the 99.9% latency of CoCaCo.

5.4. Zipfian Factors

Fig 5 displays the throughput results for different Zipfian factors in the workload whose read proportion is 80%. In terms of peak throughput, CoCaCo is nearly the same as Cassandra-Eventual and the introduced overhead is less than 4.7%; while CoCaCo greatly outperforms Cassandra-RYW and obtains 52.2%∼91.4% higher throughput. Table 4 demonstrates the corresponding $rate_{sr}$ and $depNumber_{aa}$ for different factors. The rate increases as the factor increases, but it is still small and negligibly affects the entire performance.

5.5. Dependency Number of a Writing Item

In the previous experiments, a client chooses an observed item randomly and regards it as the dependency of a writing item, which can simulate explicit causality relationships on conversations and comments in social networking.
Figure 5: Peak throughput versus Zipfian factor

Table 4: The $rate_{sr}$ and $depNumber_{aa}$ for different Zipfian factors.

<table>
<thead>
<tr>
<th>Factor</th>
<th>0.78</th>
<th>0.83</th>
<th>0.88</th>
<th>0.93</th>
<th>0.98</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rate_{sr}$</td>
<td>0.02%</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.04%</td>
<td>0.06%</td>
</tr>
<tr>
<td>$depNumber_{aa}$</td>
<td>0.045</td>
<td>0.046</td>
<td>0.056</td>
<td>0.075</td>
<td>0.103</td>
</tr>
</tbody>
</table>

Table 5: The $rate_{sr}$ and $depNumber_{aa}$ for different $depNumber_{write}$ values.

<table>
<thead>
<tr>
<th>$depNumber_{write}$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rate_{sr}$</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.03%</td>
</tr>
<tr>
<td>$depNumber_{aa}$</td>
<td>0.056</td>
<td>0.122</td>
<td>0.410</td>
<td>0.638</td>
<td>0.796</td>
</tr>
</tbody>
</table>

Service like Facebook, etc. In this experiment, we further examine how the dependency number of a writing item (denoted as $depNumber_{write}$) can affect the system performance. Note that our optimization strategy can reduce the dependency number of a writing item. Fig 6 displays the experimental results in the same workload whose read proportion is 80% under the Zipfian distribution. As shown in the figure, with the increasing of the dependency number of a writing item, the throughput slightly decreases. Table 5 demonstrates the average actual dependency number of operated items and the corresponding rates of second-round reads. The average actual dependency number $depNumber_{aa}$ increases as the dependency number of a writing item increases, while the rate basically remains unchanged. $depNumber_{aa}$ increases because a client library needs to regard more items as the dependency of a writing item. However, the rate of second-round reads basically remains unchanged, because the workload remains the same, and the possibility that an item returned by a first-round read has the timestamp that is equal or larger than the required $fts$ will not vary much.
5.6. Wide-Area Network Deployment

We deploy CoCaCo across two datacenters, and compare the peak throughput with Cassandra-Eventual and Cassandra-RYW. Each datacenter has three servers and each data item has two replicas in each datacenter. We partition 6 servers equally into 2 logical datacenters by adding network latency between two datacenters as [12] did in their experiments. The hardware of each server is the same as that in the previous experiments. The latency between the nodes in two datacenters is 19.8ms, and the bandwidth across them is not a limiting factor. Larger latency between datacenters will enlarge the gap between the curve of Cassandra-RYW and others, because Cassandra-RYW always needs to communicate with replicas in the other datacenter. Fig 7 demonstrates the experimental results, which are similar to those in LAN deployment. Under the uniform distribution, the curve of CoCaCo is close to that of Cassandra-Eventual (introduced overhead is less than 6.5%), and the throughput of them exceed 69 Kops/s for all workloads, while the throughput of Cassandra-RYW does not exceed 52 Kops/s and CoCaCo achieves a 126.7%–167.5% higher throughput than Cassandra-RYW for read-most workloads. Under the Zipfian distribution, the introduced overhead is less than 3.6%. CoCaCo also achieves a higher throughput than Cassandra-RYW, especially for read-most workloads (51.1%–75.1% higher).

6. Conclusion and Future work

Many previous causally consistent data stores did not take network partitions within each datacenter into consideration, and thus employed strong consistency models in each datacenter and guaranteed causal consistency across datacenters. In this paper, we have presented CoCaCo, a distributed
data store that can guarantee convergent causal consistency both within and across datacenters. CoCaCo adopts asynchronous replication and performs dependency checking only during handling reads, in order to achieve high availability and performance. We implemented CoCaCo based on Apache Cassandra, a production-ready data store. Our experiments demonstrated that CoCaCo introduced small overhead compared with eventual-consistent-configured Cassandra, and outperformed RYW-consistent-configured Cassandra: for read-most workloads, CoCaCo achieved 47.8%—167.5% higher throughput.

We plan to explore alternatives in the space of the system design, like employing compression techniques for dependency storage and transmission. We also plan to implement the storage layer of CoCaCo based on other production-ready data stores, and examine how the approach performs. Furthermore, we intend to provide richer semantics or new features for the CoCaCo approach, including write-read transactions support, other conflicting writes handling strategies [29, 48],
7. Acknowledgements

This work was supported partly by China 863 program (No. 2015AA01A202) and partly by National Natural Science Foundation of China (No.61370057 and No. 61421003). An early version of this work was presented as a poster in IWQoS 2016.

References


[22] Y. Tang, H. Sun, X. Wang, X. Liu, Achieving convergent causal consistency and high availability with asynchronous replication, in: Proceedings of IWQoS.


[42] Cassandra 1.0 thrift configuration.
   URL https://github.com/apache/cassandra/blob/cassandra-1.0/interface/cassandra.thrift

[43] D. K. Gifford, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears,
    Benchmarking cloud serving systems with ycsb, in: Proceedings of SoCC

[44] Cassandra 1.0 thrift configuration.
   URL https://wiki.apache.org/cassandra/ReadRepair

[45] Cassandra 1.0 thrift configuration.
   URL https://wiki.apache.org/cassandra/HintedHandoff

   URL http://wiki.apache.org/cassandra/ArchitectureInternals

[47] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding,
    J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov,
    L. Puzar, Y. Song, V. Venkataramani, Tao: Facebook’s distributed data

[48] J. J. Kistler, M. Satyanarayanan, Disconnected operation in the coda