Okapi : Causally Consistent Geo-Replication Made Faster, Cheaper and More Available

Diego Didona, Kristina Spirovska, Willy Zwaenepoel
École polytechnique fédérale de Lausanne

Abstract

Okapi is a new causally consistent geo-replicated key-value store. Okapi leverages two key design choices to achieve high performance. First, it relies on hybrid logical/physical clocks to achieve low latency even in the presence of clock skew. Second, Okapi achieves higher resource efficiency and better availability, at the expense of a slight increase in update visibility latency. To this end, Okapi implements a new stabilization protocol that uses a combination of vector and scalar clocks and makes a remote update visible when its delivery has been acknowledged by every data center.

We evaluate Okapi with different workloads on Amazon AWS, using three geographically distributed regions and 96 nodes. We compare Okapi with two recent approaches to causal consistency, Cure and GentleRain. We show that Okapi delivers up to two orders of magnitude better performance than GentleRain and that Okapi achieves up to 3.5x lower latency and a 60% reduction of the meta-data overhead with respect to Cure.

1 Introduction

Distributed data stores represent the backbone of many large-scale online services. Such data stores are often geo-replicated to improve performance, by storing a copy of the data closer to the clients, and to achieve availability, by keeping multiple copies of the data at different sites [25]. A critical decision in designing a geo-replicated store is the choice of its consistency model. At one end of the spectrum, strong consistency [15] has simple semantics, but incurs high latency and does not tolerate network partitions. At the other end, eventual consistency provides excellent performance and tolerates partitions [33], but it is hard to program with.

Causal Consistency. Causal consistency [4] hits a sweet spot in the ease of programming vs performance trade-off and has emerged as an attractive model to build geo-replicated data stores [5, 6, 8, 14, 22]. On the one hand, it avoids the long latencies and inability to tolerate network partitions of strong consistency. On the other hand, it is easier to reason about than eventual consistency and avoids some of eventual consistency anomalies.

Limitations of existing systems. At a very high level, all causally consistent systems work in the same way. Events, such as creation, reads and writes of data items, are labeled with a timestamp. This timestamp is propagated on communications between machines. New events are then always labeled with timestamps higher than the highest one received so far, thereby making sure that timestamps reflect causal order. A variety of timestamping methods have been proposed, including in particular using the current value of the physical clock of the machine on which the event occurs [5, 14]. The advantage of using physical clocks is that it is a rather concise encoding of causality, and that it is trivial to obtain.

The problem with using physical clocks for causal dependency tracking is due to clock skew between different machines. Consider, for instance, the creation of a new version of data item X on machine A, occurring at time t on the physical clock of A, and therefore labeled with timestamp t. Let this version of X be read by a client on machine B, and let that client then create a new version of data item Y. In the absence of clock skew, the value t' of B's clock at this time would be larger than t. Hence, this event can be timestamped with t', with the timestamp reflecting the causal order. However, if, due to clock skew, the value of B's clock at this time is smaller than t, then the operation needs to block until B's clock catches up to the value of t [14].

Clock skew yields similar problems also with read-only transactions, a powerful abstraction supported by the majority of the causally consistent systems [5, 6, 14, 22, 23]. The timestamp assigned to a transaction, in fact, can be higher than the value of the clock on a node involved in the transaction [5]. Recent work has shown that waits due to clock skew cause a significant reduction in

performance (up to 25% even in a small deployment [5]).

State-of-the-art systems based on physical clocks also make different trade-offs between dependency tracking overhead and the latency of transactional operations. They either use a single timestamp per item but achieve poor performance for transactional operations [12], or use a number of timestamps equal to the number of data centers to implement efficient transactions [5].

Modern applications, however, need *both* fast transactional read operations on a consistent snapshot of the data store *and* low dependency tracking overhead. Multi-key transactional reads are paramount, as they increase the expressiveness of applications [22, 24]. For example, in many services a single high-level user operation (e.g., retrieving the content of a page) translates to multiple read operations from the underlying store [25]. It is, then, highly desirable that all the retrieved values belong to a consistent causal snapshot of the data store. Moreover, small items dominate typical workloads, e.g., at Facebook [7, 25], Twitter [1] and Instagram [3]. For such workloads, dependency meta-data can easily grow bigger than the payload it refers to, with detrimental effects on scalability, communication and storage overhead.

Okapi. This paper presents Okapi, a new causally consistent geo-replicated data store. Okapi avoids the latencies caused by clock skew and implements efficient transactional reads at low dependency tracking cost.

Okapi achieves the first goal by using hybrid logical/physical clocks (HLC) to timestamp events. HLC have a physical and a logical component. Instead of waiting for the physical clock to reach a value t, a server can set the physical component of its hybrid clock to t. The logical part of the clock is, then, incremented to generate new timestamps that reflect causality among events.

To achieve the second goal, Okapi proposes a novel stabilization protocol called Universal Stable Time (UST). UST achieves higher resource efficiency and better availability, at the expense of a slight increase in update visibility latency. UST uses dependency vectors only for local updates, to efficiently serve transactional reads, and a single timestamp for replicated updates. UST provides support for higher availability during network partitions, by enforcing that data centers expose to clients only items that have been received system-wide. The (periodic and asynchronous) communication needed to check the set of remotely received items induces a slightly higher visibility latency for remote updates.

Contributions. This paper makes three contributions:

I) The design and implementation of Okapi, a causally consistent data store that i) achieves low latencies by means of a novel combination of HLC and dependency vectors; and ii) proposes a novel stabilization protocol that achieves higher efficiency and availability at the cost of a slight increase in remote updates visibility latency.

II) The exploration of Okapi's trade-offs among performance, updates visibility latency and availability.

III) The evaluation of Okapi in a large scale Amazon AWS deployment, in which we compare Okapi with Cure and GentleRain, two state-of-the-art systems that achieve causal consistency using physical clocks.

2 Definitions and System model

Causal consistency. Causal consistency requires that servers of a system return values that are consistent with the order defined by the *causality* relationship. Causality is a happens-before relationship between two events [19, 4]. For two operations a, b, we say that a causally depends on b, and write $a \rightsquigarrow b$, if and only if at least one of the following conditions holds: i) a and b are operations in a single thread of execution, and a happens before b; ii) a is a write operation, b is a read operation, and b reads the value written by a; iii) there is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

We use lower case letters, e.g., x, to refer to a key and the corresponding capital letter, e.g., X to refer to a version of the key. We say that X depends on Y if the write of X causally depends on the write of Y.

We define an item *stable* in a data center when it becomes visible to clients in that data center. An item becomes stable when all its dependencies have been received and made visible in the local data center.

We define the *visibility latency* of an item d in a data center DC as the time between the moment in which d has been created in its originating data center and the moment in which d becomes stable in DC.

Convergent conflict handling. Two operations a, b are *concurrent* if neither $a \rightsquigarrow b$ nor $b \rightsquigarrow a$. If a and b are concurrent write operations to the same key, they *conflict*. Two conflicting versions of a key can be propagated to remote replicas in different orders, potentially leading to replicas to diverge forever. Okapi implements the popular last-writer-wins rule [30] to arbitrate conflicting modifications to keys. Given two updates, the one with the highest timestamp is deterministically decided to having occurred later than the other, determining the value of the value written (possible ties are settled by looking at the id of the originating data centers of the items). Okapi can easily integrate other mechanisms to achieve state convergence, similarly to previous systems [5, 12, 14, 22].

System model. We assume a distributed key-value store that manages a large set of data items. The data-set is split into N partitions and each key is deterministically assigned to one partition according to a hash function. Each partition is replicated at M different sites, each corresponding to a different data center. Hence, a full copy

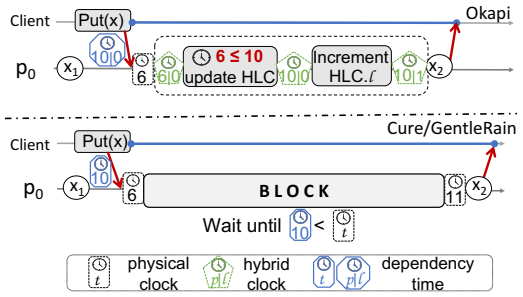


Figure 1: PUT implementation in Okapi (top) and GentleRain/Cure (bottom). The client dependency time (10) is higher than the physical clock on p_0 (6). To reflect causality, Okapi sets its HLC to $\langle 10, 1 \rangle$. GentleRain/Cure must wait until the physical clock gets to 11.

of the data is stored at each site.

We assume a multiversion data store. An update operation creates a new version of a key. Each version stores the value corresponding to the key and some meta-data to track causality. The system periodically garbage-collects old versions of keys. We further assume nodes communicate through point-to-point lossless FIFO channels.

The system supports the same programming model of the vast majority of the existing causally consistent systems, e.g., COPS [22], Orbe [12], ChainReaction [6], and Gentlerain [14], which is based on these operations:

PUT(key, val): A PUT operation assigns value *val* to an item identified by *key*. If item *key* does not exist, the system creates a new item with initial value *val*. Else, a new version storing *val* is created.

val ← GET(key): A GET operation returns the value of the item identified by *key*. A GET operation is such that its return value does not break causal consistency as explained in the following. Assume $X \rightsquigarrow Y$ and that a client c issues a GET(y) operation, receiving Y as result. Then, any subsequent GET(x) operation issued by c must return either X or a version X' such that $X' \not\rightsquigarrow X$.

$\langle$vals$\rangle$ ← RO-TX $\langle$keys$\rangle$: This operation implements a causally consistent read-only transaction [22, 23]. If a read-only transaction returns X and Y, then they are causally consistent with the issuing client's history and there is no X', such that $X \rightsquigarrow X' \rightsquigarrow Y$.

At the beginning of a session, a client c connects to a node n_c in the closest data center according to some load balancing scheme. c does not issue the next operation until it receives the reply to the current one. Operations towards data items that are not stored by n_c are transparently forwarded to the node(s) responsible for such data items, and the result is relayed back to c by n_c.

Each server is equipped with a physical clock that advances monotonically. We assume such clocks to be loosely synchronized by a time synchronization protocol, such as NTP [2]. The correctness of our protocol does not depend on the synchronization precision.

3 The Design of Okapi

We now describe the design of Okapi, focusing on its two key techniques: the use of HLC and the UST stabilization protocol. We also qualitatively compare Okapi with two state-of-the-art systems, GentleRain and Cure [1].

3.1 Using HLC to track time

Okapi uses HLC [16] to track the advancement of time, and hence to timestamp updates. A hybrid timestamp t is a tuple with a physical component $t.p$ and a logical component $t.l$. Two hybrid timestamps are compared by first comparing their physical components, and then comparing their logical components.

Each server p has a (software maintained) hybrid machine clock HLC_p and a (hardware maintained) physical clock $Clock_p$. The physical component of the HLC_p is in general different from the current value of $Clock_p$.

Each data item version stored on p has a (hybrid) update timestamp. At the time of creation of a version, its update timestamp is set to the current value of HLC_p.

Each client c has a client dependency vector DV_c, with one entry per data center. DV_c consists of hybrid timestamps that, roughly speaking, reflect the client's dependencies on data items created at each other data center.

We now show how Okapi leverages HLC to implement clock-skew resilient PUT and RO-TX operations.

PUT. When client c performs a PUT on server p, it sends its DV_c along. The server p then computes the largest element of DV_c, noted max_c, and ensures that the update timestamp of the newly created version is higher than max_c and higher than the highest update timestamp p has assigned so far. In this way, the generated timestamp reflects causality.

To this end, the server first sets $HLC_p.p$ to the maximum of $HLC_p.p$ and $Clock_p$. If $max_c < HLC_p$, then $HLC_p.l$ is incremented by one, so that the new update timestamp is higher than any previous one assigned by p. Otherwise, HLC is set to $< max_c.p, max_c.l + 1 >$, to ensure that the new update timestamp is higher than the highest dependency timestamp of the client. This ensures, without waiting, that the new update timestamp reflects causality. The server also attaches an item dependency vector to the new item. Such dependency vector is a copy of DV_c except for the entry corresponding to the local data center, which stores the timestamp of the item.

[1] Cure exposes APIs different from Okapi's and uses CRDTs [29] for state convergence. We have implemented a version of Cure that complies with the system model described in Section 2. We refer to our implementation simply as Cure.

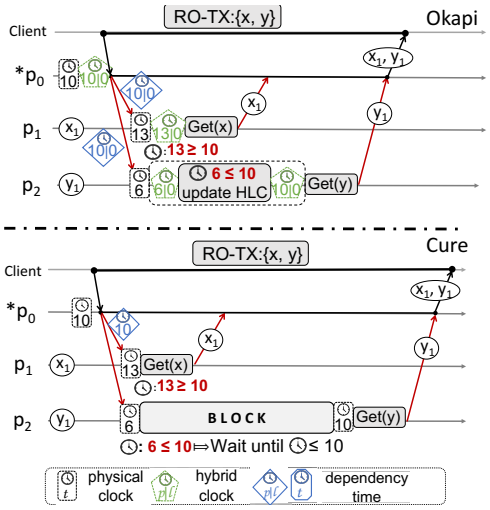


Figure 2: RO-TX implementation in Okapi (top) and Cure (bottom). The local snapshot time of the transaction (10) is higher than the value of the physical clock on p_2 (6). To avoid the creation of later items with a timestamp still ≤ 10, Okapi simply moves its *HLC* to $\langle 10,0 \rangle$. Cure needs to wait until the clock gets to 10.

In contrast, if timestamping is done using a physical clock alone, as in Cure and Gentlerain, then in the case of clock skew, the server has no other option but to wait until the physical clock catches up with max_c.

Figure 1 (top) and Figure 1 (bottom) depict the behavior of Okapi and, respectively, Cure and GentleRain when a node p with $Clock_p = 6$ receives a PUT operation from a client with $max_c = 10$.

RO-TX. The advantages of HLC are even greater for read-only transactions. In this case the transaction coordinator computes a transaction snapshot time, essentially the upper bound on timestamps corresponding to local items that are visible to the transaction [2].

The coordinator then sends requests to all the servers storing data items requested in the transaction, asking them to return the values of the versions of those data items with the largest timestamp smaller than or equal to the snapshot time.

Intuitively, when using hybrid timestamps, the server can respond immediately, regardless of clock skew, by if necessary advancing its *HLC* to the transaction snapshot time. This disallows "later" items from being created by the server with a timestamp smaller than or equal to the snapshot time, thereby preventing the transaction from "missing" any item that it should be able to access.

If, instead, physical clocks are used, then the server has no other option than to wait for the clock to catch up to the snapshot time.

The benefits of hybrid clocks are more pronounced with transactions because with physical clocks it suffices that the clock of any of the contacted servers runs behind for the waiting to occur. This easily results in a major performance impairment for systems based only on physical clocks because high-level application operations typically translate to contacting several servers at once. For example, the median number of servers contacted to retrieve a Facebook page is about 20 [25].

Figure 2 (top) and Figure 2 (bottom) compare the behavior of Okapi and Cure when serving a RO-TX with snapshot time 10 and with a contacted server p_2 whose physical clock value is 6. We only compare Okapi with Cure because the implementation of RO-TX in Cure is more efficient than in GentleRain. We shall discuss the limitations of the RO-TX implementation in GentleRain in the following section.

3.2 Efficient dependency tracking by UST

Okapi incorporates UST, a new stabilization protocol that addresses availability issues in state-of-the-art causally consistent systems. As a by-product, it considerably reduces the amount of consistency meta-data that is communicated and stored, compared to Cure, approximating that required in GentleRain. As a tradeoff, Okapi incurs a modestly higher update visibility latency.

UST in a nutshell. As with most causally consistent systems, UST allows updates originating in a data center to become visible immediately in that data center. For updates originating elsewhere, it makes them visible only when they have been replicated at all data centers.

UST works by a combination of version vectors on each server that record the latest remote updates received from their replicas in other data centers and a decentralized protocol for exchanging this information to determine what data items are fully replicated. Periodically, nodes within a data center exchange their version vectors to compute the Global Stable Vector (GSV) as the entry-wise minimum of all the version vectors in the data center. If the i-th entry of the GSV takes the value t, it means that all the servers in the data center have installed all updates originated at data center i with timestamp up to t. Periodically, peer replicas exchange their GSV to compute the Universal Stable Vector (USV), as entry-wise minimum of all the exchanged GSV. If the i-th entry of the USV takes the value t, then all updates originated at data center i have been fully replicated.

Availability. In Cure and GentleRain the failure or disconnection of a data center can cause the states of healthy data centers to diverge. Namely, it can happen that some

[2] The coordinator also determines upper bounds for the remote dependencies, as we shall discuss in the next section. We omit them here for simplicity, as they are not affected by clock skew and do not induce any waiting time in Okapi and Cure.

data items originating at that failed data center are visible in some healthy data centers but not in other ones. UST disallows this behavior by making a remote data item visible only when it has been replicated at every data center. This ensures that healthy data centers have made visible the same set of items from the failed data center and hence see the same set of remote stable dependencies even after the failure.

Meta-data overhead. UST only requires a single scalar value to be communicated and stored with a remote update to determine its visibility. In fact, when a remote update d coming from data center i arrives in data center j, all the remote dependencies of d have been already fully replicated. Hence, UST determines the visibility of d by only checking that all of d's local (i.e., of data center i) dependencies have been fully replicated. This is accomplished by simply checking if the i-th entry of the USV is lower or equal than the timestamp of d.

UST achieves the same dependency meta-data overhead for remote updates as GentleRain, but it represents a considerable improvement over Cure, which needs to store a vector of size equal to the number of data centers with each remote data item.

Unlike GentleRain, instead, UST requires that the local copy of an item d stores a dependency vector with one entry per data center (with the local entry corresponding to the timestamp of d). By this vector, Okapi can determine, at the data center granularity, the snapshot of the data store to which an item belongs. This allows Okapi to implement RO-TX efficiently by overcoming a key limitation of the design of GentleRain, which only stores the timestamps of local items.

Support for fast RO-TX. In Okapi, when receiving a RO-TX request from a client, the transaction coordinator determines the snapshot vector of the transaction. Such vector has one entry per data center and represents the freshest snapshot corresponding to stable items and including all the dependencies of the client. Every item belonging to such snapshot must be visible by the transaction. To determine whether a local item is visible to the transaction, Okapi exploits the available item's dependency vector, and checks whether it is entry-wise smaller than or equal to the snapshot vector. To determine whether a remote item d created at the i-th data center is visible to the transaction, Okapi checks if the item timestamp is lower than or equal to the i-th entry in the transaction vector. This condition is sufficient because the transaction vector includes only stable items by construction. Hence, if the condition is met, UST ensures that all of d's dependencies have already been received in the data center and are, hence, visible to the transaction.

In GentleRain, instead, the snapshot visible to a transaction is determined by a single snapshot timestamp. Every item with a timestamp lower than such value is visible to a transaction. A transaction's timestamp has to be higher than the highest dependency timestamp at the client to include all the client's dependencies. Let t be the snapshot timestamp of a transaction. To enforce that the transaction does not "miss" any item that it should be able to access, GentleRain must ensure that the local data center has received *all* items from *all* data centers with a timestamp lower than or equal to t. The duration of this synchronization step is potentially proportional to the communication delay between the local data center and the furthest data center.

Symbol	Definition
N	# partitions
M	# replicas per partition
p_n^m	The $m-$th replica of the $n-$th partition
dt_c	Dependency time at client c
GSV_n^m	Global stable vector on p_n^m
USV_n^m	Universal stable vector on p_n^m
USV_c	Universal stable vector at client c
$Clock_n^m$	Physical clock time on p_n^m
VV_n^m	Hybrid version vector of p_n^m
d	A tuple $\langle k, v, ut, sr, DV \rangle$

Table 1: Definition of symbols.

Visibility latency. The inevitable price to be paid for the increase in availability is that the visibility latency of remote updates is increased, because there needs to be communication between replicas in different data centers to compute visibility. We believe the availability gains well warrant the slight increase in update latency.

4 Protocols in Okapi

We now describe in detail the protocols run by Okapi[3]. Algorithm 1 and Algorithm 2 describe, respectively, how clients and servers implement PUT, GET and RO-TX operations. Algorithm 3 describes the management of clocks on servers. Algorithm 4 reports the UST stabilization protocol. We indicate a target client as c. At the beginning of a session, c is provided with the id m of the data center it is connected to, referred to as the local data center. We refer to the server that handles c's request as p_n^m. p_n^m can be the node with which c has established a session, or the node to which the request has been forwarded (as described in Section 2). Table 1 provides a summary of the symbols used in the discussion.

4.1 Meta-data

Item. An item d is a tuple $\langle k, v, ut, sr, DV \rangle$. k is the unique id that identifies the key of which d is a version. v is the value of d. sr is the *source replica* of d, i.e., the id of the

[3]The correctness proof is omitted for space constraint.

Algorithm 1 Okapi client c at data center m.

1: **function** GET(key k)
2: send $\langle$**GETReq** $\boldsymbol{k, USV_c}\rangle$ to server
3: receive $\langle$**GETReply** $\boldsymbol{v, USV_n^m, ut, sr}\rangle$
4: $USV_c \leftarrow max\{USV_n^m, USV_c\}$
5: **if** $(sr == m)$ **then** $dt_c = max\{dt_c,\ ut\}$ **endif**
6: **return** v
7: **end function**

8: **function** PUT(key k, value v)
9: $DV_c \leftarrow USV_c$; $DV_c[m] \leftarrow max\{dt_c, DV_c[m]\}$
10: send $\langle$**PUTReq** $\boldsymbol{k, v, DV_c}\rangle$ to server
11: receive $\langle$**PUTReply** $\boldsymbol{ut}\rangle$
12: $dt_c \leftarrow ut$ ▷ *Update client's dependency at local data center*
13: **end function**

14: **function** RO-TX(key-set χ)
▷ *Send remote dependencies (USV_n^m) and local dependencies (dt_c) info*
15: send $\langle$**RO-TX-Req** $\boldsymbol{\chi, USV_c, dt_c}\rangle$ to server $\boldsymbol{p_m^n}$
16: receive $\langle$**RO-TX-Resp** $\boldsymbol{D, USV_n^m}\rangle$
17: $USV_c \leftarrow max\{USV_n^m, USV_c\}$
18: **for** $(d \in D)$ **do**
19: read d as if it were the result of a GET ▷ *This updates dt_c if necessary*
20: **end for**
21: **end function**

data center in which d has been created. ut is the update time, i.e., the creation time of the d at its source replica. DV is a dependency vector with M entries. For a local update, $DV[i]$, $i \neq m$, is the update time of the item d' with the highest timestamp such that i) d' has originated at the $i-$th replica and ii) d depends on d'. $DV[m]$ is equal to ut. For remote updates, DV is null.

Client. A client c maintains one vector USV_c with one entry per data center. USV_c indicates the freshest stable snapshot from which any server has served a GET or a read-only transaction issued by c. c also maintains a *dependency time* dt_c, corresponding to the highest timestamp of any local item read or written by c.

Server. A server p_n^m has access to a monotonically increasing physical clock, $Clock_n^m$. p_n^m also maintains three vector clocks with M entries: VV_n^m, GSV_n^m and USV_n^m. VV_n^m is a version vector of hybrid clocks. $VV_n^m[i], i \neq m$, indicates the timestamp of the latest update/heartbeat received by p_n^m that comes from the replica at the $i-$th data center. $VV_n^m[m]$ is the version clock of p_n^m and it is used to timestamp updates. $GSV_n^m[i] = t$ means that p_n^m is aware that all the nodes in the $m-$th data center have processed all events generated in the $i-$th data center with timestamp up to t. $USV_n^m[i] = t$ indicates that p_n^m is aware that every node in every data center has installed all the updates generated in data center i whose timestamps are smaller than or equal to t. GSV_n^m and USV_n^m are read and written atomically. Okapi uses optimistic locking [21] to keep the corresponding overhead low.

4.2 Operations

GET. c sends a request $\langle$GET k, $USV_c\rangle$, where k is the key to be read. p_n^m uses USV_c to advance USV_n^m if necessary, so as to be sure to install a snapshot that is at least as fresh as the one that c has been served from so far. p_n^m then selects the version d of k with the highest timestamp such that either d is local or d's update time is smaller than or equal to the entry in USV_n^m corresponding to d's originating data center. p_n^m returns USV_n^m and d's value, timestamp and source replica to c. Upon receiving such reply, c updates USV_c and, if d is local, dt_c.

PUT. c sends a request $\langle$PUT $k, v, DV_c\rangle$, where k is the key to be written and v is the desired value to associate with k. DV_c is a dependency vector whose remote entries are equal to the ones in USV_c; the local entry is the maximum between the local entry in USV_c and dt_c. DV_c represents all dependencies established by c so far.

Upon receiving c's request, p_n^m first determines the hybrid timestamp to associate with the new update. To this end, p_n^m invokes the `updateClockOnPut` function (reported in Algorithm 3). This function advances the local

Algorithm 2 Okapi server p_n^m serving clients requests.

1: **upon** receive $\langle$**GETReq** $\boldsymbol{k, USV_c}\rangle$ from c **do**
2: $USV_n^m \leftarrow max\{USV_n^m, USV_c\}$
3: $D_k \leftarrow \{d : d.k == k\}$ ▷ *Versions chain of the desired key*
▷ *Visible version with highest timestamp*
4: $d \leftarrow argmax_{d.ut}\{D\} : (d.sr == m \vee d.ut \leq USV_n^m[d.sr])$
5: send $\langle$**GETReply** $\boldsymbol{USV_n^m, d.v, d.ut, d.sr}\rangle$ to client

6: **upon** receive $\langle$**PUTReq** $\boldsymbol{k, v, DV_c}\rangle$ from c **do**
7: *updateClockOnPut*(DV_c) ▷ *Update version vector*
8: $d.k \leftarrow k$; $d.v \leftarrow v$; $d.ut \leftarrow VV_n^m[m]$; $d.sr \leftarrow m$; $d.DV \leftarrow DV_c$
9: $d.DV[m] \leftarrow d.ut$
10: insert d in the version chain of key k
11: send $\langle$**PUTReply** $\boldsymbol{d.ut}\rangle$ to client
12: **for** $(i \leftarrow 0 \ldots M, i \neq m)$ **do**
13: send $\langle$***Replicate*** **d.k, d.v, d.ut**$\rangle$ to p_n^i
14: **end for**
15: lastOutMsg $\leftarrow Clock_n^m$

16: **upon** receive $\langle$**RO-TXReq** $\boldsymbol{\chi, USV_c, dt_c}\rangle$ from c **do**
17: *updateClockOnTx*(dt_c)
18: $USV_n^m \leftarrow max\{USV_c, USV_n^m\}$ ▷ *Install newer USV if needed*
19: $lts \leftarrow VV_n^m[m]$ ▷ *Take freshest local snapshot*
20: $\chi_i \leftarrow \{k \in \chi : partition(k) == i\}$ ▷ *Set of requested keys per node*
21: $D \leftarrow \emptyset$ ▷ *Items to return to client*
22: **for** $(i\ s.t.\ \chi_i \neq \emptyset)$ **do** ▷ *Done in parallel*
23: send $\langle$***SliceREQ*** $\boldsymbol{\chi_i, lts, USV_n^m}\rangle$ to p_i^m
24: receive $\langle$***SliceRESP*** $\boldsymbol{D_i}\rangle$ from p_i^m
25: $D \leftarrow D \cup D_i$
26: **end for**
27: reply $\langle$$\boldsymbol{D, USV_n^m}\rangle$ to c

28: **upon** receive $\langle$**SliceREQ** $\boldsymbol{\chi, lts, USV_i^m}\rangle$ from the coordinator p_i^m **do**
29: *updateClockOnTx*(lts) ▷ *Update $Clock_n^m$ to cope with clock skew.*
30: $USV_n^m \leftarrow max\{USV_i^m, USV_n^m\}$ ▷ *Install newer USV if needed*
31: $TS \leftarrow USV_i^m$; $TS[m] \leftarrow lts$ ▷ *Transaction snapshot vector*
32: $D \leftarrow \emptyset$
33: **for** $k \in \chi$ **do**
34: $D_k \leftarrow \{d : d.k == k \wedge ((d.sr == m \wedge d.DV \leq TS) \vee$
35: $(d.sr \neq m \wedge d.ut \leq TS[d.sr]))\}$
36: $D \leftarrow D \cup argmax_{d.ut}\{D_k\}$ ▷ *Freshest visible version*
37: **end for**
38: reply $\langle$***SliceRESP*** $\boldsymbol{D}\rangle$ to p_i^m

39: **upon** receive $\langle$**Replicate** $\boldsymbol{k, v, ut}\rangle$ from p_n^i **do**
40: create new item d
41: $d.k \leftarrow k$; $d.v \leftarrow v$; $d.ut \leftarrow ut$; $d.sr \leftarrow i$
42: insert d in the version chain of key $d.k$
43: $VV_n^m[i] \leftarrow d.ut$

Algorithm 3 Okapi server p_n^m: clock management.

```
1: function UPDATECLOCKONPUT (DV_c)
2:    hd ← max{DV_c}                                  ▷ Find highest dependency
3:    max_p ← max{VV_n^m[m].p, Clock_n^m, hd.p}       ▷ Max physical clock
4:    if (max_p == VV_n^m[m].p == DV_c[m].p) then     ▷ Local phys clock behind
5:       l = max{VV_n^m[m].l, DV_c[m].l} + 1
6:    else if (max_p == VV_n^m[m].p) then l = VV_n^m[m].l + 1
7:    else if (max_p == DV_c[m].p) then l = DV_c[m].l + 1
8:    else l = 0                                      ▷ Local phys clock higher than dependency
9:    end if
10:   VV[m].p ← max_p; VV[m].l ← l
11: end function

12: function UPDATECLOCKONTX(ts)
13:   if (ts > VV[m] ∧ ts > Clock_n^m) then VV[m] ← ts endif
14: end function

15: function UPDATECLOCKONHEARTBEAT
16:   if (Clock_n^m > VV_n^m[m].p) then
17:      VV_n^m[m].p ← Clock_n^m; VV_n^m[m].l ← 0
18:   end if
19: end function

20: upon every Δ time do
21:   if Clock_n^m ≥ lastOutMsg+Δ then
22:      updateClockOnHeartbeat()
23:      for each server p_n^j, j ∈ {0...M−1}, k ≠ m do
24:         send ⟨HEARTBEAT VV_n^m[m]⟩ to p_n^j
25:      end for
26:   end if
27:   lastOutMsg ← Clock_n^m

28: upon receiving ⟨HEARTBEAT ct⟩ from p_n^j do
29:   VV_n^m[j]←ct
```

Algorithm 4 Okapi server p_n^m: GSV and USV computation.

```
1: upon every Δ_G time do
2:    GSV_n^m[j] ← min{VV_i^m[j]}, ∀j = 0...M−1, ∀i = 0...N−1

3: upon every Δ_U time do
4:    V[j] ← min{GSV_n^i[j]}, ∀j = 0...M−1, ∀i = 0...N−1
5:    USV_n^m ← max{V, USV_n^m}                       ▷ Enforce monotonicity of USV
```

entry of the version clock of p_n^m, $VV_n^m[m]$, with a hybrid timestamp that is higher than the highest entry in DV_c and than the current version clock $VV_n^m[m]$. Then, p_n^m creates a new version d of k, and replies to c with d's timestamp. This is used by c to update dt_c. Finally, p_n^m replicates d by sending to its replicas a copy of d, except $d.DV$.

Upon receiving such replication message, a replica p_n^i inserts a copy of d in the version chain corresponding to $d.k$ and sets $VV_n^i[m] = d.ut$.

RO-TX. c sends a request $\langle RO-TXReq, \chi, USV_c, dt_c \rangle$, where χ is the set of keys to be read. USV_c and dt_c are provided so that the transaction is served from a snapshot that includes c's dependencies.

Upon receiving c's request, p_n^m acts as the coordinator for the corresponding transaction. First, p_n^m computes the local transaction snapshot time, lts. This time represents the highest timestamp of local items visible to the transaction. lts is computed as the maximum between the local clock at the coordinator and dt_c. p_n^m also updates USV_n^m if necessary. In this way, the snapshot defined by USV_n^m and lts is the freshest snapshot that includes all the dependencies established by c. p_n^m sends USV_n^m and lts to every node p_i^m that holds at least one key in χ, together with the set of keys to be read.

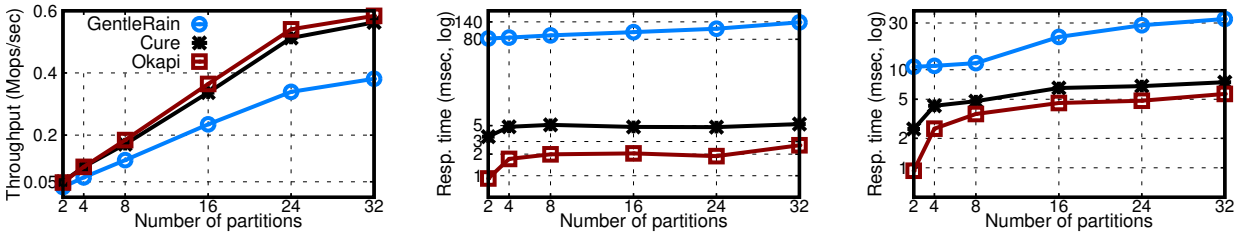
Upon receiving such message, p_i^m invokes the `updateClockOnTx` function (reported in Algorithm 3). This function advances $VV_i^m[m]$ in case it is lower than lts. p_i^m also updates its USV_i^m if it is smaller than the one proposed by the coordinator. Then, p_i^m computes the transaction's snapshot vector TV starting from the USV and lts proposed by p_n^m. TV is equal to USV_n^m in the remote entries. The local entry is, instead, the local transaction timestamp proposed by p_n^m. For each key to be read, p_i^m determines the version d with the highest timestamp such that d is visible to c according to TV.

A local item is visible if its update time and its dependencies fall within the boundaries defined by TV (Line 34). A remote item is visible if its update time falls within the boundaries of TV (Line 35). Since TV is computed starting from a USV, this condition implies that the remote item is also stable.

The set of all read items is sent back to p_n^m. Upon collecting all such replies, p_n^m forwards them back to c together with USV_n^m. Finally, c updates USV_n^m and, for each item in the returned set, updates its dependency meta-data as when processing the result of a GET.

Heartbeats. If p_n^m does not receive update requests from clients, it does not send replication messages to its replicas either. Therefore, other replicas cannot increase the m-th entry in their version vector, and the $m-$th entry of the USV cannot advance. To avoid this scenario, a partition that does not receive updates for a period of time longer than Δ, broadcasts its latest local hybrid version clock time to its replicas. The function `UpdateClockOnHeartbeat` computes the heartbeat timestamps by advancing the local version clock $VV_n^m[m]$ to $Clock_n^m$ if $Clock_n^m$ is higher than $VV_n^m[m]$. Heartbeat messages and update replication messages are sent (and received) in order of increasing update timestamps and clock values. Upon receiving a heartbeat with timestamp t from p_n^m, p_n^i sets $VV_n^i[m] = t$.

Stabilization protocol. Every Δ_G time units, partitions within a data center exchange their version vectors. p_n^m computes GSV_n^m as the aggregate minimum of known version vectors. Similarly to previous work [14, 5], Okapi organizes nodes within a data center as a tree to reduce message exchange. Every Δ_U time units, replicas at different partitions exchange their GSV and compute the USV as the aggregate minimum of the received GSV. Because USV_n^m is also updated when serving client requests, it can happen that USV_n^m becomes greater than GSV_n^m in some entries. Thus, p_n^m enforces that entries in USV_n^m are monotonically increasing.



(a) Throughput scalability. (b) RO-TX avg. resp. time (log). (c) PUT avg. resp. time (log).

Figure 3: Performance with increasing scales of the system. Clients perform a RO-TX involving two partitions and a PUT on a random partition. Okapi achieves better or similar peak throughput with respect to GentleRain and Cure but considerably lower latencies. This is because, thanks to HLC, Okapi never blocks when serving an operation.

Garbage collection. Servers within a data center periodically exchange the transaction snapshot vector corresponding to their oldest active transactions and compute the garbage collection vector GV as the entry-wise minimum of those vectors. If no transaction is active on p_n^m, p_n^m sends a fake transaction snapshot vector, as computed in Algorithm 2 Line 31. A server retains every version of any key k it stores up to and including the freshest version that would be visible to a transaction with transaction vector GV. Older versions are removed. That is, Okapi retains up to the oldest version of k that could still potentially be visible to a transaction.

5 Evaluation

5.1 Methodology and performance metrics

We evaluate Okapi by responding to these questions:

- How well does Okapi scale?
- What throughput can Okapi achieve?
- What latencies do Okapi operations achieve?
- How much does Okapi benefit from HLC?
- What is the penalty in update visibility latency incurred by Okapi to support higher availability?
- What is the communication overhead of UST?

We answer these questions by comparing Okapi with Cure and GentleRain on a large scale public cloud infrastructure. We evaluate the performance of these systems with different workloads and deployment settings. We report achievable throughput and average operation latencies, with a focus on PUT and RO-TX operations, since GET operations are not affected by clock skew. We also report remote updates visibility latency, communication costs and dependency tracking meta-data overhead. We conduct our evaluation using a benchmark that allows us to accurately assess the sensitivity of Okapi to key workload characteristics like the number of partitions involved in a transaction and write intensity.

5.2 Experimental test-bed

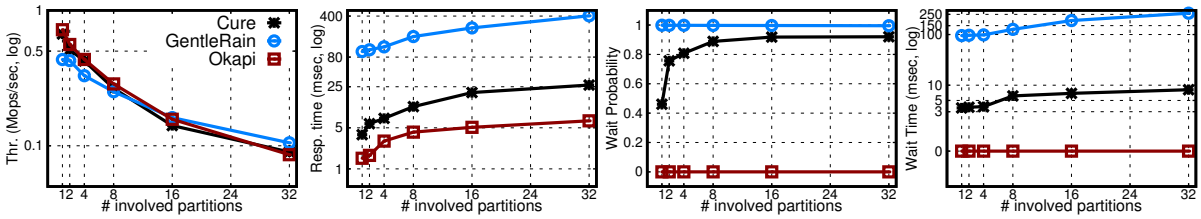
We consider an Amazon AWS deployment with 3 data centers and 32 partitions each. The data centers are in Oregon, N.Virginia and Ireland. We use $c4.large$ instances (2 virtual CPUs and 3.75 GB of RAM). Data is stored in-memory, without any fault tolerance mechanism. This allows us to evaluate Okapi without taking into account the dynamics and overhead of logging/replication. Okapi can be extended to achieve fault tolerance by means of standard techniques [26, 20, 32].

Each partition is composed of one million key-value pairs. We consider small items, with keys and values of 8 bytes, as representative of many production workloads [1, 3, 7, 25]. Keys are chosen within each partition according to a zipf distribution with parameter 0.99. Clients are collocated with servers, establish their sessions with the collocated server and perform operations in closed loop. We run NTP [2] to synchronize physical clocks. As in previous work [5], clocks are synchronized before each experiment. We use the NTP server 0.amazon.pool.ntp.org. All the stabilization protocols are run every 5 milliseconds. Heartbeats are sent by a node if it does not serve any put request for 1 millisecond.

Hybrid timestamps, similarly to physical ones, are encoded with 64 bits. If a node has to increase the logical part of its HLC but $HLC.l$ has already reached the maximum value, the node has to resort to waiting. We use the 48 most significant bits of a HLC as physical part and the other 16 as logical part. As such, a hybrid timestamp can track phyisical time up to microsecond granularity and can encode up to 2^{16} logical events [16]. With this settings, we have never witnessed a node resort to waiting.

5.3 Experimental results

Scalability. We evaluate the scalability of Okapi by running a workload on an increasing number of partitions, from 2 to 32. In this workload, each client performs a RO-TX involving two partitions (so as to keep the num-



(a) Throughput (log). (b) RO-TX avg. resp. time (log). (c) RO-TX wait probability. (d) RO-TX avg. wait time (log).

Figure 4: Performance while varying the number of partitions involved in a transaction. Clients perform a RO-TX and a PUT touching random partitions. Okapi achieves better or comparable peak throughput with respect to GentleRain and Cure, but achieves considerably lower latencies. By means of HLC and UST, Okapi never blocks a transaction. Instead, Cure stalls transactions because of clock skew among nodes in the *local* data center. GentleRain incurs the highest waiting time and probability because it has to wait to receive *all* the items from *all the remote* data centers that are included in the snapshot visible to a transaction.

ber of contacted partitions fixed regardless of the scale) and a PUT. The partitions touched by the operations are chosen uniformly at random. Figure 3 depicts the result of the experiment, reporting peak throughput in Figure 3a, average response time of the RO-TX operation in Figure 3b and of the PUT operation in Figure 3c.

Okapi achieves 50% higher throughput than GentleRain and a slightly higher throughput than Cure. Okapi, however, achieves much lower latencies for both RO-TX and PUT operations, up to two orders of magnitude lower than GentleRain and 100% lower than Cure.

Okapi achieves this result by never blocking operations. Cure and GentleRain need to activate many more clients than Okapi to compensate for the idle waiting times and saturate their resources. Okapi and Cure use vector clocks, so their peak throughput is similar. Okapi's throughput is slightly higher because UST allows for better resource efficiency. The use of scalar clocks leads GentleRain to incur very long waiting times to serve a transaction, as explained in Section 3.2. The excessive number of client threads, needed to fill the long waiting times, limits GentleRain's overall scalability.

Sensitivity to RO-TX characteristics. We now evaluate the performance of Okapi when serving transactions that span different numbers of partitions. To this end, we consider a workload in which clients issue a RO-TX to read p keys and then write one key belonging to a random partition. Each read key is stored on a different partition, and partitions are chosen uniformly at random. We fix the number of partitions per data center to 32 and we analyze the performance of the three systems while varying p from 1 to 32.

Figure 4a shows the throughput achieved by the considered systems. Figure 4b depicts the average transaction response time corresponding to the throughput values of Figure 4a. Figure 4c and Figure 4d report, respectively, the probability that a transaction is stalled before being served and the duration of the stall. Figure 5 reports, for different values of p, the average RO-TX response time as a function of the throughput.

The plots show that Okapi achieves a slightly better throughput than Cure, for any value of p. Okapi is up to 60% better than GentleRain for transactions that span up to 8 partitions. Then GentleRain achieves a marginally higher throughput than Okapi. Cure and GentleRain, however, incur considerably higher latencies because of their blocking behavior, for any value of throughput. GentleRain achieves a higher throughput when the number of contacted partitions is high because it timestamps transactions with a scalar and not with a vector, as in Okapi and Cure. This results into a lower utilization of the network, which enables more concurrency when a transaction involves many partitions.

The plots also show the different blocking behaviors of Cure and GentleRain. In Cure, the probability of waiting

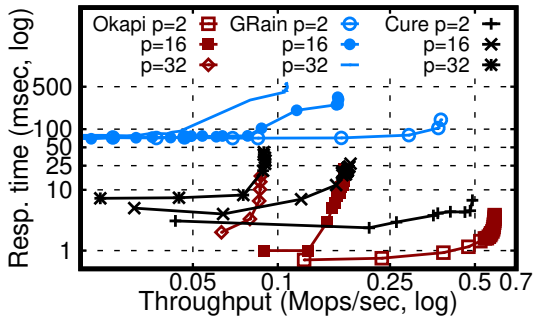
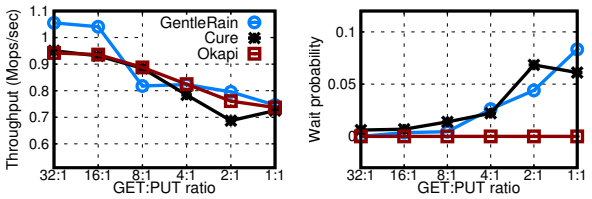


Figure 5: RO-TX avg. resp. time (log) as a function of the throughput and of the # partitions involved in a transaction (p). Okapi achieves the lowest latency and almost always the highest throughput. For $p = 32$ Okapi attains a slightly lower throughput than GentleRain, but achieves a 2 orders of magnitude lower latency.



(a) Throughput. (b) PUT wait probability.

Figure 6: Performance of transaction-less workloads with different GET:PUT ratios (32 partitions). Okapi never blocks PUT operations and thus performs slightly better than Cure. GentleRain achieves slightly higher throughput in read-dominated scenarios because it only uses scalar dependency timestamps instead of vectors. Okapi trades this marginal penalty for much bigger gains in RO-TX latencies and higher availability.

due to clock skew increases with the number of contacted partitions. The waiting time is proportional to the clock skew, and it is in the order of 5-10 milliseconds on average. GentleRain always waits to receive from all the data centers all the items that are in the transaction snapshot. The waiting time is, hence, mainly proportional to the communication latency with the furthest data center.

Sensitivity to write intensity. We now analyze the sensitivity of the three systems to the workload write intensity. To this end, we run different workloads consisting of only GET and PUT operations, using 32 partitions. Clients read g items on g distinct partitions chosen uniformly at random and then update one item on a random partition. We vary g from 1 to 32. Figure 6a reports peak throughput and Figure 6b reports the probability that a PUT operation is stalled due to clock skew.

The plots show that Okapi achieves a higher throughput than Cure. The difference between the two increases as the probability of stalling a PUT due to the clock skew increase with the write intensity of the workload. Okapi is, instead, comparable or competitive with GentleRain. In the most read-intensive workloads Okapi incurs a slight throughput penalty ($< 10\%$) because of the use of vector clocks instead of a single scalar. We believe this small cost is well worth the huge improvement that Okapi attains in the RO-TX implementation and the higher level of availability that Okapi achieves.

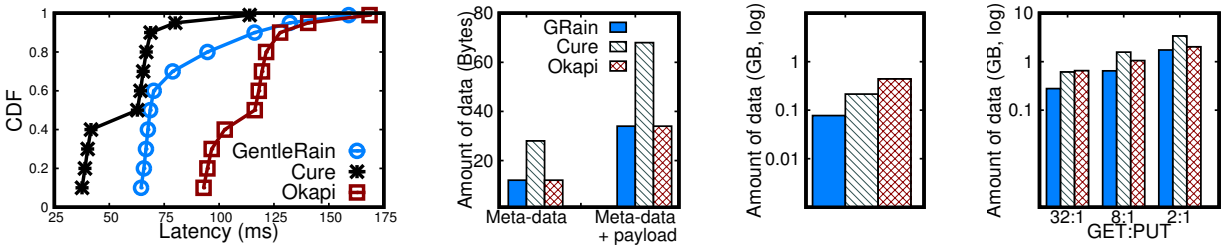
Implications of UST. We now evaluate the effects of the stabilization protocols of the three systems. Figure 7a reports the CDF corresponding to the visibility latencies of remote updates in the 32:1 GET:PUT workload on 2 partitions. Figure 7b depicts the amount of data replicated per update. Figure 7c reports the amount of data exchanged to execute the stabilization protocols and Figure 7d reports the total amount of data exchanged among nodes (for the stabilization protocol and updates replication) while varying the write intensity of the GET-PUT workload on 32 partitions.

Okapi achieves the highest remote update visibility latency, for the sake of higher availability, followed by Cure and GentleRain. In Cure, the visibility latency in data center DC_R of an item d originated in DC_L depends on the delay between DC_L and DC_R [5]. In GentleRain, instead, the visibility latency depends on the delay between DC_R and its furthest data center [12]. The CDF of Okapi and Cure is bi-modal (one mode per remote data center) because the visibility latency of d depends (also) on the communication latency between DC_L and DC_R. GentleRain's CDF, conversely, is unimodal because the remote update visibility latency depends on the communication delay between DC_R and its furthest data center.

Deferring the visibility of updates allows Okapi to match the resource efficiency of GentleRain when replicating updates. Okapi and GentleRain need only 12 bytes of meta-data, corresponding to the source replica (4 bytes) and update time (8 bytes). Cure needs additional 8 bytes for each remote entry of the dependency vector. In our setting, then, Cure requires 28 bytes of meta-data per update, which is more than two times the overhead incurred by Okapi and GentleRain. In our experiments, an update contains additional 16 bytes to encode the key and the value. Even if the payload amortizes the meta-data overhead, the amount of data sent by Cure to replicate an update is still almost 60% higher than in Okapi and GenleRain. In Okapi and GenleRain dependency meta-data for replicated updates is insensitive to the scale of the system. In Cure, instead, it grows linearly with the number of data centers in the system.

UST requires an additional round of inter-data center communication to achieve higher availability. For this reason, the stabilization protocol of Okapi is more expensive than Cure's and GentleRain's. Such overhead, however, is compensated for by the reduced meta-data overhead achieved by Okapi. If we consider, in fact, the total amount of data exchanged, i.e., stabilization protocol overhead and replication cost, Okapi incurs a communication overhead similar to Cure in read dominated workloads, and lower than Cure as the write intensity increases. GentleRain's stabilization protocol is the most network efficient regardless of the write intensity of the workload. Its gains against UST, however, decrease as the write intensity increases and the dominant communication cost becomes the updates replication.

Okapi could significantly reduce the UST overhead by piggybacking the computation of the USV to the one of the GSV. We have not experimented with this design yet.



(a) Remote updates visibility latency.

(b) Bytes/replicated update.

(c) Data exchanged for the stabilization protocol (log).

(d) Data exchanged for the stabilization protocol and updates replication (log).

Figure 7: Effects of UST. UST incurs a slightly higher visibility latency than Cure and GentleRain to support higher availability (a). As a by-product, UST matches the remote updates dependency tracking overhead of GentleRain, which only uses scalar clocks (b). Okapi's stabilization protocol exchange more data than Cure's and GentleRain's to achieve higher availability (c). This overhead is amortized by the reduction in meta-data for replicated updates (d).

6 Related Work

Our work is primarily related to the literature on causally consistent systems. The first breed of such systems includes Bayou [27], lazy replication [18], ISIS [10], causal memory [4], and PRACTI [9]. They implement causal consistency but assume single-machine replicas and do not consider partitioned data-sets. COPS [22] represents the first in a new class of systems, which implement causal consistency for both replicated and partitioned data stores. This second set of systems includes Eiger [23], Bolt-on causal consistency [8], ChainReaction [6], Orbe [12], GentleRain [14], SwiftCloud [34] and Cure [5]. Okapi differs from these systems on two levels: event timestamping and dependency tracking.

Event timestamping. COPS, Eiger, ChainReaction, Bolt-on and Orbe use logical clocks to timestamp items. These systems exchange explicit dependency check messages among partitions to verify that a remote update can be made locally visible. GentleRain and Cure, instead, use loosely synchronized physical clocks and implement a stabilization protocol to determine the visibility of remote updates. GentleRain and Cure achieve higher performance than previous systems but incur additional synchronization delays to cope with clock skew. By employing HLC [16], Okapi implements a cheap stabilization protocol and is insensitive to clock skew. Concurrently to our work, the use of HLC to achieve causal consistency has also been investigated in GentleRain+ [28]. GentleRain+ simply augments GentleRain with HLC to make PUT operations robust against clock skew. The stabilization protocol and the implementation of transactions are the same as in GentleRain, so GentleRain+ inherits the limitations of GentleRain that we have described in the paper. Conversely, Okapi uses a novel combination of HLC and dependency vectors to implement efficient transactions. As we have shown, this combination is paramount to achieve scalability and low-latency for production-like workloads, which rely on efficient snapshot reads. Moreover, Okapi achieves higher availability than GentleRain+ thanks to UST.

Dependency tracking. The systems based on logical clocks keep detailed dependency information, encoded as a dependency list [22, 23, 6, 8] or matrix [12]. The techniques proposed to reduce the resulting overhead have downsides like per-update acknowledgement messages among replicas [12], call-backs to the client [12], or delay the visibility of updates also in the local data center [13, 34]. GentleRain and Cure track dependencies at a coarser granularity. GentleRain uses a single timestamp to achieve minimal overhead but incurs high waiting times to serve read-only transactions. Cure uses dependency vectors to avoid this issue but incurs a dependency tracking overhead linear in the number of data centers. Okapi uses dependency vectors too but reduces the meta-data for remote updates at the cost of slightly delaying their visibility at remote sites.

Okapi's design is also related to the use of physical and hybrid clocks in systems that target different consistency guarantees, e.g., Spanner [11], Clock-SI [12], PhysiCS-NMSI [31] and CockRoachDB [17].

7 Conclusion

We have presented Okapi, a novel geo-replicated key-value store that achieves causal consistency. Okapi uses hybrid logical/physical clocks and a novel stabilization protocol to achieve better performance, resource utilization and availability than existing approaches.

References

[1] How much text versus metadata is in a tweet? http://goo.gl/EBFIFs.

[2] NTP: The network time protocol. http://www.ntp.org.

[3] Storing hundreds of millions of simple key-value pairs in redis. http://goo.gl/ieeU17.

[4] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995), 37–49.

[5] AKKOORATH, D. D., TOMSIC, A., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIÇA, N., AND SHAPIRO, M. Cure: Strong semantics meets high availability and low latency. In *Proc. of ICDCS* (2016).

[6] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of EuroSys* (2013).

[7] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proc. of SIGMETRICS* (2012).

[8] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on causal consistency. In *Proc. of SIGMOD* (2013).

[9] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. Practi replication. In *Proc. of NSDI* (2006).

[10] BIRMAN, K. P., AND JOSEPH, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Jan. 1987), 47–76.

[11] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013), 8:1–8:22.

[12] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proc. of SoCC* (2013).

[13] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. Closing the performance gap between causal consistency and eventual consistency. In *Proc. of PaPeC* (2014).

[14] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proc. of SoCC* (2014).

[15] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[16] KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical physical clocks. In *Proc. of OPODIS* (2014).

[17] LABS, C. Cockroachdb. an open source, survivable, strongly consistent, scale-out sql database. https://www.cockroachlabs.com.

[18] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 360–391.

[19] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[20] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[21] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. of NSDI* (2014).

[22] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. of SOSP* (2011).

[23] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proc. of NSDI* (2013).

[24] LU, H., HODSDON, C., NGO, K., MU, S., AND LLOYD, W. The snow theorem and latency-optimal read-only transactions. In *In Proc. of OSDI* (2016).

[25] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proc. of NSDI* (2013).

[26] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC* (1988).

[27] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible update propagation for weakly consistent replication. In *Proc. of SOSP* (1997).

[28] ROOHITAVAF, M., AND KULKARNI, S. S. Gentlerain+: Making gentlerain robust on clock anomalies. *CoRR abs/1612.05205* (2016).

[29] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Proc. of SSS* (2011).

[30] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.

[31] TOMSIC, A. Z., CRAIN, T., AND SHAPIRO, M. Physics-nmsi: Efficient consistent snapshots for scalable snapshot isolation. In *Proc. of PaPoC* (2016).

[32] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proc. of OSDI* (2004).

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

[34] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write fast, read in the past: Causal consistency for client-side applications. In *Proc. of Middleware* (2015).