Practical and Fast Causal Consistent Partial Geo-Replication
[*]

Pedro Fouto
NOVA LINCS & DI
FCT / NOVA University of Lisbon
Lisbon, Portugal
p.fouto@campus.fct.unl.pt

João Leitão
NOVA LINCS & DI
FCT / NOVA University of Lisbon
Lisbon, Portugal
jc.leitao@fct.unl.pt

Nuno Preguiça
NOVA LINCS & DI
FCT / NOVA University of Lisbon
Lisbon, Portugal
nmp@fct.unl.pt

Abstract

Distributed storage systems are a fundamental component of large-scale Internet services. To keep up with the increasing expectations of users regarding availability and latency, the design of data storage systems has evolved to achieve these properties by exploiting techniques such as partial replication, geo-replication, and weaker consistency models. How to combine all these techniques in a single solution in a practical and efficient way is highly challenging.

In this paper we propose a novel replication scheme that can offer causal+ consistency in a geo-distributed scenario with partial replication, where datacenters replicate different portions of the entire database. We leverage on a recently proposed methodology that decouples the propagation of data and causality-tracking metadata. Our solution presents a novel causal consistency tracking and enforcing algorithm, focusing on maximizing parallelism in the execution of remote operations which, as we show, has a significant influence on the performance of a partially replicated system. We also propose and implement a design to integrate our solution in the popular Cassandra database. Experimental results show that, by exploring a new position in the trade-off between throughput and data visibility (by balancing the execution of local and remote operations, respectively), our solution presents overall good performance.

Keywords: Causal Consistency, Partial Replication, Geo-Replication

[*]The work presented here was partially supported by the Lightkone European H2020 Project (under grant number 732505) and NOVA LINCS (through the FC&T grant UID/CEC/04516/2013). Parts of this report have been published in IEEE NCA'18.

1 Introduction

Distributed data storage solutions [1,2,13,17–19,21] have become key components of many large-scale Internet services, such as commercial platforms, social networks, among others. To ensure fast response times to clients, these systems typically have a geo-replicated architecture, where data is replicated across servers scattered in multiple datacenters across the world. This allows clients to contact the closest datacenter, avoiding the penalty of long round trip times to reach remote datacenters (i.e, in other continents). The performance of storage solutions significantly affects the overall performance of services that use them, in particular regarding user-perceived latency. The increasing need for high availability and fast response times has lead many geo-replicated storage systems to favor replication protocols offering weak consistency models instead of strong consistency [2,13,18,19,21].

However, resorting to weak consistency models, such as eventual consistency, puts a high burden on application developers, that have to explicitly deal with many consistency anomalies allowed by these models. This has lead to the emergence of geo-replicated storage systems that provide causal+ consistency [2,6,9,15,26,27], which has been demonstrated to be the strongest consistency model that is compatible with allowing client operations to execute locally without the need of coordinating with remote replicas [3,24] hence, ensuring high availability even in faulty scenarios where entire datacenters become temporarily unavailable.

Unfortunately, most existing solutions offering causal+ consistency in geo-replicated scenarios rely on total replication, where each datacenter holds a full copy of the data managed by the storage system. The increasing number of datacenters and the recent interest in Internet-of-Things (IoT) and Internet-of-Everything (IoE) applications, that produce large volumes of data, has sparked the interest in exploring geo-replicated systems that provide partial replication, where data objects are grouped in partitions that are replicated only at a small sub-set of all datacenters. This significantly reduces the storage and communication overhead of full replication schemes.

While many recent systems have implemented causal replication models, they do so using different techniques which result in each implementation having a different behavior. When comparing these behaviors, the main trade-off that can be observed is between data freshness (how long an update takes to be seen by users connected to each replica) and throughput [14,23]. This trade-off is caused by the mechanisms employed by these systems to track causality, with some systems trying to reduce the amount of metadata used by compressing it [6,18] and hence losing precision leading to false dependencies, which sacrifices data freshness. Other solutions rely on additional metadata [9,11], sacrificing throughput (and potentially latency). Furthermore, these systems were not designed to take advantage of partial replication and cannot be easily adapted to such purpose without requiring partial replicas to handle metadata associated with items that they do not replicate.

In this paper we set out to devise a practical partial geo-replicated storage system that finds the sweet spot between throughput (and user perceived latency) and data freshness in a way that is fault-tolerant. To do so, we depart from a technique consisting on decoupling the propagation of metadata to track causality from the propagation of update operations between datacenters. Our work presents two main contributions:

First, we present a novel algorithm for tracking and enforcing causal consistency, which explores a new position in the trade-off between throughput and data visibility. Instead of focusing on executing operations locally as fast as possible (at the cost of higher visibility times), we balance the execution between local and remote operations, by using just enough metadata to allow concurrency in the execution of remote operations. This allows us to have an increase of performance in partially replicated scenarios, where clients need to migrate between datacenters, thus requiring low data visibility times.

Second, we present a design for integrating our solution with the well known Cassandra storage system [21]. For achieving this, a number of challenges needed to be addressed, such as enforcing causal consistency in the existing quorum-based replication protocol and guaranteeing that causality is enforced when replicating across datacenters.

Our experimental results, obtained on a real cloud setting composed of nine datacenters scattered throughout the world, show that our solution achieves a better trade-off between throughput and client observed latency and data-freshness when compared with current state-of-the-art solutions providing causal+ consistency in a partial geo-replicated system, and with an acceptable overhead in relation to configurations of Cassandra that only offer eventual consistency.

The remainder of this paper is organized as follows: Section 2 discusses related work; Section 3 presents our proposal; The integration details of our solution with Cassandra are presented in Section 4 and the experimental work in Section 5; Finally, Section 6 concludes the paper.

2 Related Work

Consistency in cloud storage systems A large number of geo-replicated cloud storage systems have been developed in the last decade, with different designs that explore trade-offs between consistency and availability. Some systems [8,16] adopt a strong consistency model, where the execution of all operations is serialized. This requires replica coordination for executing an operation, leading to high latency and potentially to low availability in the presence of failures.

Other systems [2,13,21,26] adopt weak consistency models, where an operation executes in a single replica, being later asynchronously propagated to other replicas. These systems exhibit low latency and high availability, but concurrent updates, may lead to (temporary) replica divergence. Under eventual consistency [13,21], systems only guarantee that replicas will eventually converge. Under causal consistency [2,26], the system guarantees that an update is only visible after all updates that it may depend upon are visible.

Causal Consistency with Total Replication Lamport [22] introduced the happens-before relation among events in a distributed system. This relation is the base for causal consistency, where an update must be delivered after all updates that happened-before it. ISIS [5] and lazy replication [20] provide a causal communication primitive, where messages are delivered in causal order. When used for maintaining replicated state in multiple replicas, they enforce causal consistency. These systems track causal dependency using version vectors, where each entry in the vector corresponds to a replica.

In modern cloud infrastructures, where the data storage is composed by thousands of machines located in several datacenters, tracking causality relying on version vectors with one entry per machine is unpractical. COPS [26] was the first system to introduce the concept of causal+ consistency, a consistency model that combines causal consistency guarantees with the convergence offered by eventual consistency [25]. COPS assumes that each datacenter offers linearizability and it tracks causal dependencies by recording for each write the version of the last written object and of all read objects since the last write. These dependencies are used to decide when a write can be applied in a datacenter. Eiger [27] is an evolution of the COPS system that uses a similar approach for enforcing causal+ consistency, while supporting both read-only and write-only transactions. ChainReaction [2] is a geo-distributed key-value data storage system that relies on a variant of chain-replication for providing causal+ consistency. Inside a datacenter, it uses the same information as COPS for tracking causality, minimizing the information that needs to be recorded by delaying write operation until their dependencies are stable. For enforcing causal consistency across datacenters, it uses a version vector with one entry per datacenter and bloom filters to enable efficient remote execution.

Orbe [11] enforces causal consistency by using two-dimensional dependency matrices to track dependencies in a client session. Cure [9] enforces causal consistency while supporting generic transactions. The dependencies of each operation are recorded in a version vector with one entry per datacenter. To allow replicas to synchronize in a peer-to-peer fashion, while guaranteeing that reads do not block, Cure includes a stabilization mechanism that computes which remote updates can be made visible. GentleRain [18] enforces causal consistency by recording dependencies as a single scalar. When compared with Cure, this approach is more efficient but leads to more false dependencies and delays in the execution of remote operations.

Eunomia [15] allows local client operations to always progress without blocking by relying on a intra-DC stabilization technique. This technique relies on a hybrid (combinations of physical and logical) clock per data partition that is associated with each client operation. Based on these clocks, the Eunomia service computes a total order of operations that captures all causality relations established per datacenter. This total order is then employed by the Eunomia service to execute remote operations on each datacenter. The use of the local datacenter stabilization procedure can delay the propagation of operations to remote datacenters significantly and hence, affect negatively the global visibility times of operations.

Causal Consistency with Partial Replication Saturn [6] is the system that most resembles our proposal. Saturn is a causality tracking metadata service that propagates the identifiers of every operation using a global dissemination tree interconnecting all datacenters. As operations identifiers are propagated according to a FIFO order in each link, they reach each datacenter in causal order – this is the order in which remote operations are executed in a datacenter. Saturn uses an optimization algorithm for computing the dissemination tree to minimize the delay in the propagation of updates. When a node of the tree fails, a new tree needs to be recomputed, which is a heavy (time consuming) process. Using a tree also creates additional challenges. First, the tree is a bottleneck of the system, as every identifier must be propagated through the tree. Second, propagating information between two nodes may involve several intermediate nodes, increasing latency. Our work avoids these limitations by adopting an alternative approach, where information is propagated directly between every pair of nodes, which requires slightly more causality tracking metadata. Additionally, we propose an algorithm for optimizing the concurrent execution of remote operations, which must be serialized in Saturn.

Causal consistency in cloud databases As mentioned before, a large number of cloud databases providing weak consistency models have been developed in recent years. Among the databases used in production, such as Dynamo, Cassandra and Riak, none provides causal consistency. Bailis et. al. [4] proposed a mechanism for allowing clients to access a causally consistent database. The proposed approach relies on a layer between the client and the datastore that intercepts client operations and extends them with the necessary information to track causal dependencies. Freitas et. al [12] adopted a similar approach for enforcing causal consistency (or less restrictive session guarantees [10]). When compared with our solution, these approaches have the advantage of requiring no change in the database servers. However, they require storing additional information on the database.

3 Design

In this work we propose a new partial replication scheme, C^3, that enforces causal+ consistency. In this section we present the generic algorithms used in our system, and in the next section we detail how to integrate these algorithms in Cassandra, a popular geo-replicated cloud database.

3.1 System Model

C^3 is designed to extend an existing storage system by integrating our replication scheme to enforce causal+ consistency. Thus, our design assumes a set of properties from the underlying storage system that are commonly implemented by current production solutions.

Storage system consistency: We assume that the storage system provides eventual consistency across datacenters, allowing an update to execute in a datacenter without coordinating with other datacenters, i.e., updates are propagated asynchronously to other datacenters.

Within a datacenter, we assume that after a write completes, all following reads will return the written value. While a write of an object is in progress, the value returned by a read can be either the old or the new value. Unlike linearizability, it is possible that after a read returns the new value, some other read (issued by other client) may still return the old value. This is the common behavior of replication systems based in quorums that return the most recently written value (and not the value of the majority).

Partial geo-replication: We assume that the underlying storage system supports (some form of) partial geo-replication. This means that each datacenter knows where each data object is replicated, i.e., it is possible to know where an update needs to be propagated to. While some systems natively support this feature (e.g., Cassandra), it could also be added to other systems easily by restricting the replication set for each object.

Sharding: We assume that inside each datacenter, objects are sharded across multiple nodes. This is not a correctness requirement for our approach but, it is necessary to ensure that performance benefits can be obtained from increasing concurrency inside a datacenter.

3.2 Architecture

Our design separates the system in two layers: the causality layer and the datastore layer. Clients interact with the datastore layer for executing their operations. The datastore layer is responsible for executing operations in the local datacenter, and propagating them to remote datacenters. It coordinates with the causality layer to decide when an operation can be executed in order to enforce causal consistency. The causality layer is responsible for propagating causality tracking information among operations. In our system, this information consists of *labels* – for each write operation, our system generates a *label* consisting of an unique identifier and the causal dependencies for the operation. The causal dependencies are a vector with one entry per datacenter.

This separation between the datastore and causality layers brings relevant benefits: *(i)* it is possible to isolate the logic for enforcing causality in the causality layer, allowing this approach to be independent of any existing datastore system, and making it possible to use multiple datastore systems to materialize the datastore layer; *(ii)* the causality layer only needs to store and process very small pieces of data which allows it to be very efficient.

Each datacenter contains a causality layer instance which is responsible for managing the causality information inside a datacenter and exchanging causality-related information – labels – with other causality layer instances. Each causality layer instance communicates, propagating labels, directly with every other causality layer instance. This approach simplifies fault-tolerance, as a fault in one datacenter does not affect the communications among the remaining datacenters.

3.3 API

Our system design handles three types of operations: *(i) read* operations, to read the state of the database; *(ii) write* operations, to modify the state of the database; and *(iii) migrate* operations, to change the home datacenter of a client.

Algorithm 1: C^3: Causality Layer Algorithm

```
 1: Local State:
 2:    Π //Represents the causality layer processes in the system (one per dc)
 2:    LocalDC //Name of this node's datacenter
 3:    WriteCounter //Counter used to assign identifiers to write operations (unique when combined with LocalDC)
 4:    ExecutingClock //Tracks executing writes.  ExecutingClock[DC]→HighestExecutingWriteId
 5:    ExecutedClock //Tracks completed writes.  ExecutedClock[DC]→HighestCompletedWriteId
 6:    WaitingOps //Stores labels to be executed.  WaitingOps[DC]→Queue(labels).  Messages in each queue are ordered by their identifier
 7:    CurrentWrites //Keeps track of writes being executed in the local dc
 8:    AheadExecutedOps //Tracks operations that finished executing out of order

 9: Upon Init ( ) do:
10:    LocalDC ⟵ $getLocalDatacenter()$ //Name of this node's datacenter
11:    WriteCounter ⟵ 0
12:    Foreach $p \in \Pi$ do:
13:       dc ⟵ GetDatacenter(p)
14:       $ExecutingClock[dc]$ ⟵ 0
15:       $ExecutedClock[dc]$ ⟵ 0
16:       $WaitingOps[dc]$ ⟵ {}
17:       $AheadExecutedOps[dc]$ ⟵ {}
18:    $CurrentWrites$ ⟵ {}

19: Upon Receive (ClientMigrateLbl, s, possibleDCs) do: //Migration label from client/datastore
20:    lblDeps ⟵ ExecutingClock
21:    targetDC ⟵ GetDatacenterWithLowestLoad(possibleDCs)
22:    Send (CausalityMigrateLbl, GetCausalityNodeInDatacenter(targetDC), LocalDC, lblDeps, s)

23: Upon Receive (DatastoreWriteLbl, s, writeInfo, targets) do: //Write label from datastore
24:    WriteCounter ⟵ WriteCounter+1
25:    lblId ⟵ WriteCounter
26:    lblDeps ⟵ ExecutingClock
27:    Foreach $p \in \Pi$ do:
28:       dc ⟵ GetDatacenter(p)
29:       If ($dc \in targets$) then
30:          Send (CausalityWriteLbl, p, LocalDC, lblDeps, lblId, writeInfo[dc], targets[dc])
31:       Else
32:          Send (CausalityClockLbl, p, LocalDC, lblDeps, lblId)

33: Upon Receive (CausalityMigrateLbl, s, sourceDC, lblDeps, respondTo) do:
34:    enqueue(WaitingOps[sourceDC], {CausalityMigrateLbl, sourceDC, lblDeps, respondTo})
35:    Trigger CheckWaitingOps()

36: Upon Receive (CausalityWriteLbl, s, sourceDC, lblDeps, lblId, writeInfo, localTargets) do:
37:    enqueue( WaitingOps[sourceDC],
                {CausalityWriteLbl, sourceDC, lblDeps, lblId, writeInfo, localTargets})
38:    Trigger CheckWaitingOps()

39: Upon Receive (CausalityClockLbl, s, sourceDC, lblDeps, lblId) do:
40:    enqueue(WaitingOps[sourceDC], {CausalityClockLbl, sourceDC, lblDeps, lblId})
41:    Trigger CheckWaitingOps()

42: Procedure CheckWaitingOps() //Inefficient but simpler to understand
43:    executed ⟵ true
44:    While (executed) do:
45:       executed ⟵ false
46:       Foreach $queue \in WaitingOps$ do:
47:          {lblType, sourceDC, lblDeps, ...} ⟵ peek(queue)
48:          //Check if all positions in lblDeps are = or < to ExecutedClock
49:          If SmallerOrEqual(lblDeps, ExecutedClock) then
50:             executed ⟵ true
51:             lblType match:
52:                case CausalityMigrateLbl → Trigger ExecuteMigration(dequeue(queue))
53:                case CausalityClockLbl → Trigger ExecuteClock(dequeue(queue))
54:                case CausalityWriteLbl → Trigger ExecuteWrite(dequeue(queue))

55: Procedure ExecuteMigration(CausalityMigrateLbl, sourceDc, lblDeps, respondTo)
56:    Send (MigrateResponse, respondTo)

57: Procedure ExecuteClock(CausalityClockLbl, sourceDC, lblDeps, lblId)
58:    ExecutingClock[sourceDC] ⟵ ExecutingClock[sourceDC]+1 /Will always equal labelId
59:    Trigger FinishedExecuting(sourceDC, lblId) //Nothing to execute, simply update clocks

60: Procedure ExecuteWrite(CausalityWriteLbl, sourceDC, lblDeps, lblId, writeInfo, localTargets)
61:    ExecutingClock[sourceDC] ⟵ ExecutingClock[sourceDC]+1 //Will always equal labelId
62:    CurrentWrites[writeInfo] ⟵ {sourceDc, lblId, localTargets}
63:    Foreach $target \in localTargets$ do:
64:       Send (ExecuteWrite, target, writeInfo[target]) //Instructs datastore to execute the write

65: //Acknowledge of writes completion from datastore nodes
66: Upon Receive (WriteAcknowledgeMessage, s, writeInfo) do:
67:    //Check if the write was executed in a quorum of nodes
68:    If CheckIfWriteCompleted(CurrentWrites(writeInfo)) then
69:       {sourceDc, lblId, localTargets} ⟵ CurrentWrites[writeInfo]
70:       CurrentWrites[writeInfo] ⟵ $null$
71:       Trigger FinishedExecuting(sourceDC, lblId)

72: Procedure FinishedExecuting(sourceDc, lblId)
73:    If ExecutedClock[sourceDC]+1 $\neq$ lblId then //Previous write(s) are still executing
74:       //Store until previous write(s) complete
75:       AheadExecutedOps[sourceDC] ⟵ AheadExecutedOps[souceDC] $\cup$ {lblId}
76:    Else
77:       ExecutedClock[sourceDC] ⟵ lblId
78:       //Check if subsequent operations already completed
79:       ExecutedClock[sourceDC] ⟵ CheckAheadOps(AheadExecutedOps[sourceDC])
80:       Trigger CheckWaitingOps()
```

As our system is designed to extend an existing storage system, it can use any of the existing read and write operations available in the underlying storage system.

At each moment, a client has a home datacenter. A client can only issue read and write operations for data objects that are replicated in its home datacenter. Thus, prior to executing an operation over an object that is not replicated in its home datacenter, the client must execute a migration. Migrate operations do not need to be explicitly issued by applications, as the client-side library used to issue operations should detect when they are needed and issue them automatically.

3.4 Enforcing causality

The goal of the causality layer is to maintain the necessary information to guarantee that any operation execution respects causal consistency. As we assume that in general there are more read operations than write operations [7], our design favors the execution of read operations. As such, we allow read operations to complete directly in the datastore, without any coordination with the causality layer.

Under our system model, in tandem with systems that use quorum-based replication, while a write is in progress, the value returned by a read might be either the old or the new value. For enforcing causal consistency we need to guarantee that: *(i)* for a write, $w(o)$, after a client returns a version that reflects $w(o)$ it cannot later read an older version that does not reflects that write yet; *(ii)* for two writes on the same or different objects, $w_1(o_1)$ and $w_2(o_2)$, where w_1 causally precedes w_2, if a client reads a version of object o_2 that reflects w_2, all following o_1 reads must return a version reflecting w_1.

The first property could be enforced immediately by a system providing one-copy serializability (or linearizability). However, our system model assumes a weaker consistency model within a datacenter. As such, we can enforce this property in two ways: by caching the values returned by read operations (in the client side) or; by always contacting the same quorum of nodes when reading a given object.

For the second property, we record information about the dependencies of each operation, and then guarantee that operations that depend on each other always execute in the correct order. We rely on the causality layer to implement this strategy, and present its pseudo-code in Alg. 1. This layer maintains the following information: an *operation counter*, used to timestamp operations; an *executed clock*, a vector with one entry per datacenter recording the timestamp of the latest operation executed from that datacenter; an *executing clock*, a vector recording the timestamp of the latest operation from each datacenter in execution in the local datacenter (Alg. 1, lines $3-5$). Note that *executing clock* $\geq$ *executed clock*.

When the application issues a write, the client sends the operation to the datastore, which forwards the information about the operation to the causality layer and propagates the operation itself to the datastore nodes that are responsible for executing the write. The operation is not immediately executed in any of the datastore nodes.

When the causality layer receives the information about a new local operation (Alg. 1, line 23), including a unique identifier and the set of datacenters where the operation is to be delivered to, it increments the local operation counter, uses it to assign a timestamp to the operation, and sets the operation dependencies to be those captured by the executing clock – this information is named the operation *label*. We note that this is necessary in our system model, as for operations that are being currently executed, it is not possible to determine if the client has read any of the values being written. Thus, the only safe approach is to assume that the write depends on all writes that are in progress.

This label is then propagated to the causality layer in every datacenter that replicates the modified object (Alg. 1, lines $27-32$). Whenever the causality layer of a datacenter receives a label, it adds it to a log of pending operations (Alg. 1, line 36). A pending operation is ready to execute when the operations it depends upon have already completed, i.e., all entries of the *executed clock* are larger or equal to the entries in the dependencies of the operation (Alg. 1, lines $42 - 54$). When an operation is ready to execute, the causality layer notifies the local nodes that are responsible for executing the operation that they can execute the operation (Alg. 1, lines $60-64$). These nodes acknowledge the causality layer when the execution completes, so that the causality layer knows when to update its *executed clock* (Alg. 1, lines $66 - 71$).

This approach guarantees that an operation only executes after all operations it depends upon have completed, while also allowing parallel execution of concurrent operations.

Migrate

When reading or writing an object that is not replicated in the local datacenter it is necessary to guarantee that the system still enforces causal consistency. Intuitively, what is necessary is to guarantee that the operation of accessing the remote replica executes after all operations that have been observed by the client. As the client does not record any information about the operations it depends on (i.e, the client's causal history), this information needs to be obtained from the causality layer. Again, the safe approach is to assume that the client might have observed a version of the database that reflects all operations currently being executed.

To guarantee this, we implemented a migrate operation that allows a client to change its home datacenter. This operation executes similarly to a write operation, with the difference that it is only propagated to the target (new home) datacenter (Alg. 1, lines $19 - 22$), and that when the operation is ready to execute in the target datacenter, the client is notified that the migration has completed (Alg. 1, lines $55 - 56$). After the migration completes, a client can send its operations directly to the new home datacenter.

3.5 Discussion

Our approach relies on a safe tracking of causal dependencies, while still allowing a high degree of concurrency as we discuss in this section.

When compared with solutions that track causal dependencies precisely, such as COPS [26], our approach has the advantage of requiring no additional information to be stored in the storage system. To track dependencies precisely, these systems require, at least, a version identifier to be recorded with every object version.

When compared with systems that also include a causality layer to control the execution of operations, such as Saturn [6] and Eunomia [15], our system relies on a weaker consistency model inside a datacenter, allows increased concurrency, requires no additional information to be stored with each object and does not require the client to manage any information about dependencies.

Both Saturn and Eunomia assume that each storage system implements linearizability inside a datacenter. Although linearizability can be implemented efficiently, current data storage systems that use quorum-based replication often adopt a weaker model – e.g. Cassandra and Riak with any quorum specification and MongoDB with majority read concern.

Regarding concurrency, in Saturn, as writes executed concurrently in a datacenter by multiple clients are serialized, all information about the fact that they are concurrent is lost. Thus, when executing these operations in remote datacenters, they need to execute serially, i.e., an operation from a datacenter can only start executing after the previous operation from the same datacenter completes. On the contrary, in our system, as writes include their dependencies, they can execute concurrently. We note that the trade-off is that, in our system, a write can only execute in a datacenter after the completion of the writes that are being executed when the write starts. However, if multiple writes are issued concurrently, they all have to wait for the completion of these writes, but all can be executed concurrently afterwards.

4 Integration with Cassandra

We now explain how we integrated our solution with Cassandra[1]. From a conceptual point of view, we need to: *(i)* guarantee that Cassandra behaves according to our system model; *(ii)* modify the execution of operations to integrate our replication scheme.

To guarantee that Cassandra conforms our system model, we need to guarantee that after a write completes in a datacenter, all following reads observe a database version that reflects the completed write. This can be achieved by choosing the Cassandra's *LOCAL_QUORUM* consistency level in all operations. After a write completes in a local quorum, by the properties of quorums, it is guaranteed that any subsequent local quorum intercepts the previous one, thus always returning the written value. To avoid rare situations where a client observes a value being written and, in a subsequent operation. observes a previous version of that same data object, we resort to maintaining, at the client side, a cache of recently accessed data objects. This cache can be purged whenever the client executes a write or migration operations.

To integrate our replication scheme, we needed to make some changes to the write execution flow. In Cassandra, a write starts with the client sending the operation to the to-be coordinator node, which then forwards the operation to the relevant nodes, and waits for a response from a configurable subset of those nodes before responding to the client.

To this flow, we made the following modifications: *(i)* the coordinator, additionally, sends the information about the write to the causality layer, as soon as it propagates the operation to the relevant nodes; *(ii)* after a node receives a write operation, it only executes it after both the operation data and the label have been received. After executing an operation, the node acknowledges that execution to the causality layer.

Cassandra also employs a mechanism called *read repair*. This mechanism is used to update replicas that might have not been updated in prior writes (as messages are lost and the coordinator only waits for a quorum of nodes). Since we do not want this read repair mechanism to violate causality, by being executed across datacenters and applying updates to nodes before the label arriving, we modified it by restricting its operation to within each datacenter.

5 Evaluation

In this section, we present our evaluation and subsequent results. We start by presenting our implementation of Saturn, which we used in our experiments, followed by the description of the experimental environment and results.

5.1 Saturn

In addition to implementing our solution prototype, and in order to be able to compare it with Saturn, we implemented a version of it on top of Cassandra. Since Saturn requires changing the behavior of both the datastore layer and the client, instead of just intercepting messages and generating labels, we needed to further modify Cassandra.

Following the design presented in [6], we leverage on the Cassandra's coordinator to materialize the *frontend*, the Saturn's component that abstracts the data store internals from the client. This was done because the coordinator already mediates the access of the client to the datacenter internals. To materialize Saturn's *gears*, the component responsible for generating and manipulating Saturn's operation

[1]In our prototype, we have used Cassandra commit: `https://github.com/apache/cassandra/commit/ec7d5f9da63b60475e9ed76987d632bd7410ed11`.

	EUS	**JAP**	**AS**	**AUS**	**IND**	**CA**	**WUS**	**EU**	**BR**
EastUS	-	154	221	206	190	27	88	133	122
Japan	155	-	72	121	129	169	106	235	263
Asia	221	72	-	95	62	234	172	187	331
Australia	206	121	96	-	152	226	185	287	314
India	190	128	63	153	-	202	228	128	298
Canada	27	169	234	225	203	-	67	96	133
WestUS	87	106	172	181	228	65	-	157	188
Europe	133	235	186	287	127	96	157	-	192
Brazil	122	262	330	314	298	134	188	192	-

Table 1: Latency between experimental evaluation datacenters (ms)

labels[2], we use Cassandras' nodes by changing their behavior to generate a label for each received write operation. These labels are generated taking in account the client's label (which is attached to the write operation) and are then propagated to the causality layer. Moreover, the write operation mutation was modified to write the label to the data store with the modified data, which is essential to ensure that the label can be sent to clients that read the object in the future.

To implement the *label sink* and *remote proxy* components of Saturn, the causality layer components that are responsible for receiving and delivering labels from and to the datastore, we modified our own causality layer to implement the Saturn logic. All our changes were performed to try to strictly respect the logic and algorithms described in [6]. Still, we note that we have not enforced linearizability inside each datacenter, relying instead in a weaker consistency model. The implication of this is that our implementation is more efficient than a correct Saturn implementation at the cost of not correctly enforcing causal consistency in all situations.

5.2 Experimental Environment

In our experiments we used the Microsoft Azure cloud platform. Since we are particularly interested in geo-replicated settings, we used multiple datacenters (9 in total) spread across the world. Table 1 shows the locations of the datacenters used and the measured latency between them. Since we want to study sharding effects, we need multiple Cassandra nodes in each datacenter. We created 4 virtual machines per datacenter, each running a Cassandra instance. The machines are of the type *A2 v2*, each having 2 CPU cores and 4GB of RAM.

To model the partial replication, we created 9 data partitions, one associated with each datacenter (which acts as the main datacenter for that partition) and replicated each across multiple datacenters. The number of datacenters which replicates each partition varies between 3 and 5. Each partition is replicated in its main datacenter, at least one nearby datacenter, and then in one or more random datacenters. Inside each datacenter, each partition is also replicated in 3 out of the 4 existing nodes. Table 2 summarizes the distributions of partitions across datacenters.

YCSB In our experiments, we used *Yahoo! Cloud System Benchmark* [7] to emulate multiple clients executing operations in the system and to gather performance metrics for these experiments. In our experiments, clients are attached to a primary datacenter (the datacenter that is closest to them) and can operate in two modes. In the first, that we named *local only* clients only access objects that are replicated in their home datacenter. In the *remote* mode, clients execute some operations in their home datacenter, and then migrate to a remote datacenter, execute some operations there, and finally migrate

[2]Note that Saturn's labels are not the same as our solution's labels, their purpose, however, is similar.

Partition	EUS	JAP	AS	AUS	IND	CA	WUS	EU	BR
EastUS	x					x			x
Japan		x	x	x	x			x	
Asia		x	x	x	x				x
Australia	x	x		x					
India			x		x			x	
Canada	x					x	x	x	x
WestUS	x		x		x	x	x		
Europe			x			x	x	x	
Brazil	x							x	x

Table 2: Partition distribution across datacenters for the experimental evaluation

back to their (previous) home datacenter (where they continue this cycle). To this end we have adapted YCSB to support the migrate operation. We also added some extra logic to handle when a client should migrate and how many operations it should execute in each datacenter. These changes are only relevant when running experiments using either our solution or Saturn, as there is no notion of migrate operation in Cassandra. Since Saturn assumes the existence of a client library that manages a label associated with the last operation of each client, we also had to adapt YCSB to emulate this behavior of the client library, by handling the logic of maintaining, updating, and attaching the client label to operations.

Experimental Parameters To study how our solution compares to others, we ran experiments using four different baselines, where three of them are configurations of the Cassandra datastore the offer different consistency properties (all of them within the scope of eventual consistency) and the other our implementation of Saturn:
Cassandra with local quorum consistency (*E-LQ*): This configuration uses a regular Cassandra cluster, with eventual consistency where clients execute operations locally using the consistency level *LOCAL_QUORUM*, which means they only need to wait for the response of a quorum of local nodes (i.e two out of three). When the client needs to execute operations over remote data, it uses the consistency level *TWO* which, in the large majority of cases, results in a quorum composed of nodes located in the closest remote datacenter. This approach ensures that all operations executed by clients are processed by the single closest datacenter. This configuration is relevant as it allows us to measure the overhead of our solution in relation to a baseline that gives very few consistency guarantees.
Cassandra with quorum (*E-Q*): When using this configuration clients use Cassandra's consistency level *QUORUM* for all their operations, which means they need to wait for a response from a quorum (majority) of the *total* number of nodes that replicate the data object manipulated by the operation.
Cassandra with each-quorum (*E-EQ*): In this configuration clients use the consistency level *EACH_QUORUM* for write operations, meaning they need to wait for a quorum (2 nodes) in *each* datacenter where the target data object is replicated before responding to the client. Just like in the *E-LQ* configuration, clients use the consistency levels *LOCAL_QUORUM* and *TWO* for local and remote reads, respectively.

In the *E-Q* and *E-EQ* configurations, write quorums always intercept read quorums. These configurations are relevant as they attempt to get closer to causal consistency guarantees.
Saturn (*C-SAT*): This configuration uses our implementation of Saturn (whose details were explained before).

When using either of the solutions providing causality (our solution and Saturn), one of the machines in each datacenter needs to be running the causality layer process. Since we do not want that machine to be unbalanced in terms of load, we decrease the number of Cassandra tokens in that machine, which reduces the amount of data it replicates, thus decreasing its load. We ran experiments with several values and found out that 100 tokens in the machine that runs the causality layer and 256 (the default) tokens in the others is adequate.

For the Saturn configuration, we attempted to run a centralized algorithm developed by the authors that would generate the optimal tree for our setup, however, due to the high number of datacenters in our setup, the algorithm would take too much time to complete (multiple weeks). As such, we generated a tree ourselves which attempted to minimize the overall latency between all (neighboring) datacenters.

As for the client configuration, we created 4 extra virtual machines in each datacenter, with the same specifications as the ones that run the datastore. We then run an instance of YCSB in each of these machines with a variable number of client threads which ranges from 50 to 350, in steps of 50 (in total this varies the total number of clients between $1,800$ and $12,600$). The number of operations executed by each YCSB instance is always $25,000$ (for a total of $900,000$ operations), which is then divided by the number of client threads running. Each client executes an equal number of read and write operations, following a zipfian distribution for selecting the object that is targeted by each individual operation.

When using the remote operation mode for clients, the number of local and remote operations executed are decided by generating Poisson-distributed random numbers, using the lambda values of 95 and 5, respectively, resulting in 95% of local operations and 5% of remote operations. These values assume that storage systems are partitioned in a way that most data will be available in the client's local datacenter.

5.3 Results

We now present and discuss our experimental results. The main metrics studied are system throughput, client perceived latency, and data visibility times. We start by presenting the experiments comparing our work (C^3) with Cassandra's configurations that allow us to measure the overhead introduced by enforcing causal consistency. These results are not directly comparable with our solution, as all Cassandra configurations offer consistency guarantees strictly weaker than our solution. We then compare our system with Saturn.

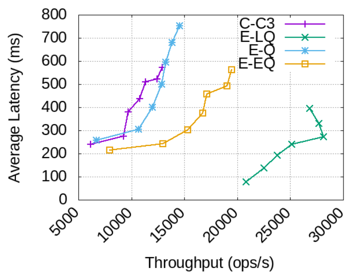
5.3.1 Performance vs multiple Cassandra configurations

Figures 1a and 1d report the relation between the number of clients executing operations simultaneously and the overall throughput of the system and client perceived latency when comparing our solution with different Cassandra configurations. Each point in each line represents increasing number of clients executing operations simultaneously and it goes from 1800 (first point in each line) to 12600 (last point in each line). Better results are represented by lower and rightmost points, since these represent lower latency and higher throughput values, respectively. Figure 1a represents clients executing only local operations while Figure 1d represents both local and remote operations.

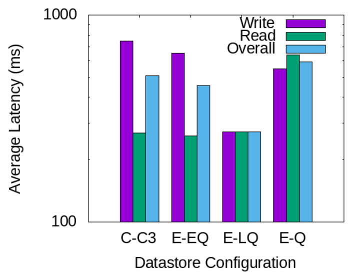
The results are not surprising, with the best results visible when the client only needs a response from the local datacenter (E-LQ), and worsening as more and further nodes need to be contacted (E-Q and then E-EQ). As expected, due to the overhead of providing causal consistency guarantees, our system shows the lowest performance.

Figures 1b and 1c report the average and 95 percentile latency of operations as perceived by clients while using only local operations while Figures 1e and 1f show the same results regarding experiments with both local and remote operations. These figures report the results of the experiments with 9000 clients (corresponding to the fifth point in Figures 1a and 1d). We selected this data point because there is significant load but the system is not saturated, achieving maximum throughput (for both our solution and Saturn, as seen in Figures 2a and 2d). Note that the Y axis has a logarithmic scale to make the plots more readable.

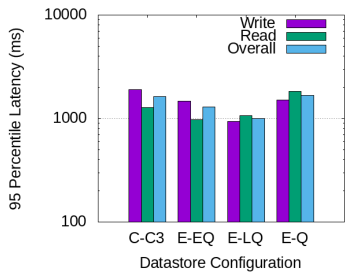
Local read latencies are very similar across all cases, except for the configuration of Cassandra using (global) quorums (Fig. 1b). This makes sense since in all other four experiments the client reads from



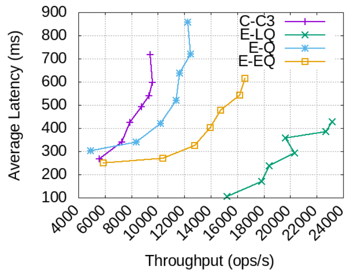
(a) Throughput and latency with only local operations



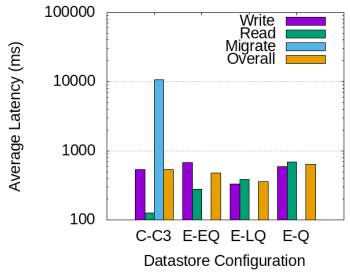
(b) Average latency with only local operations



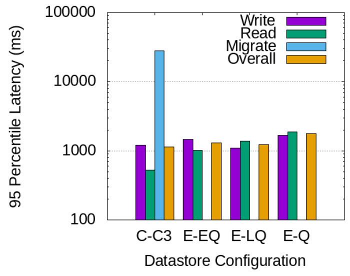
(c) 95 percentile latency with only local operations



(d) Throughput and latency using local and remote operations



(e) Average latency with local and remote operations



(f) 95 percentile latency with local and remote operations

Figure 1: Performance comparison between C^3 and multiple Cassandra configurations

a local quorum without any coordination in the causal systems (including in our solution), while in this particular one it must read from a global quorum. In write operations, the Cassandra configurations have latency proportional to the number and distance of nodes needed for gathering the quorum, leading to the expected result that the local quorum has lower latency, followed by quorum and then each quorum. Our solution has a higher write latency, not because of the nodes needed to be contacted but because every write operation needs to coordinate through the causality layer.

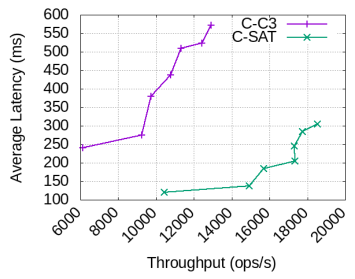
Regarding the latency of both local and remote operations (Fig. 1e), Cassandra results are similar to the previous ones since the execution of remote operations follows the same logic as local operations, the only difference being the latency between the client and the remote datacenter. Our solution's behavior, however has changed considerably since now the client needs to use migrations before moving to another datacenter. Overall, this means that read latency in our solution will be smaller, since the datastore nodes have less load, but migrate operations take a long time to complete, as they need to wait for all causally related write operations to be applied on the remote datacenter.

5.3.2 Performance vs Saturn

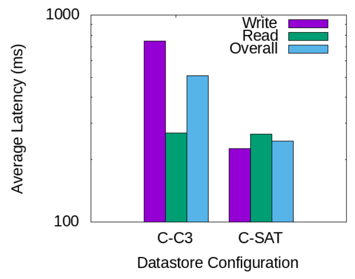
Figures 2a to 2f represent the same measurements as Figures 1a to 1f with the difference being that we are now comparing our solution with Saturn.

Starting with the throughput vs latency plots, in the local only scenario (Fig. 2a) Saturn appears to have better performance than C^3. This happens because Saturn executes local write operations faster, without coordinating with the causality layer. Also, the slow execution of remote operations leaves more resources available to execute local operations, by sacrificing data visibility.

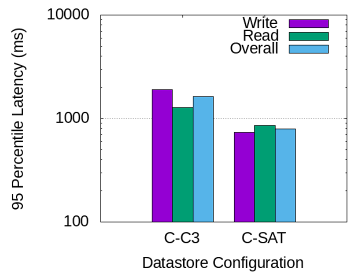
In the scenario with remote operations (Fig. 2d), C^3 shows much better performance than Saturn. It is worth pointing out that in a partially replicated setting, where migrations are unavoidable, this scenario



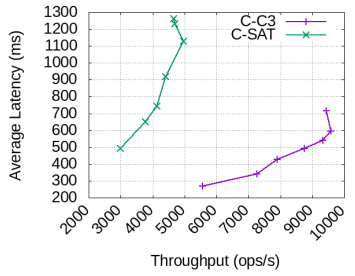
(a) Throughput and latency with only local operations



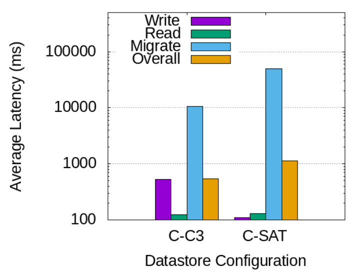
(b) Average latency with only local operations



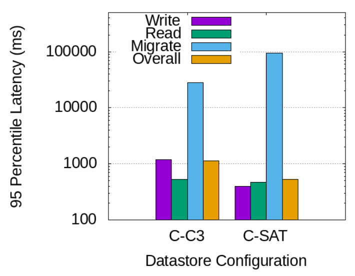
(c) 95 percentile latency with only local operations



(d) Throughput and latency using local and remote operations



(e) Average latency with local and remote operations



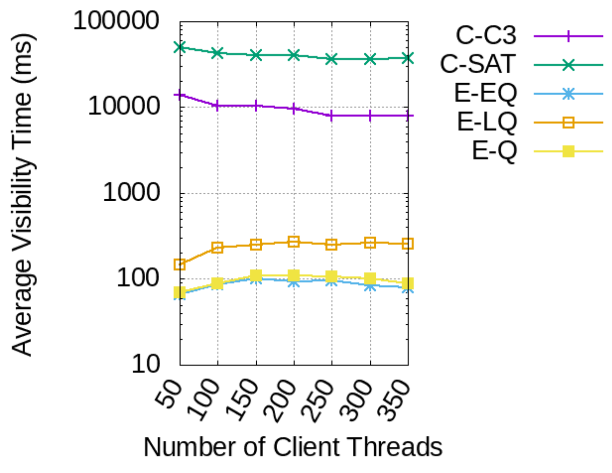
(f) 95 percentile latency with local and remote operations

Figure 2: Performance comparison between C^3 and Saturn

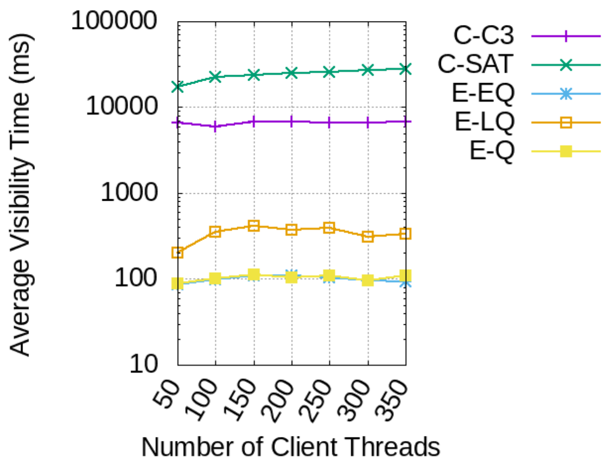
is more realistic than the local only scenario. In this scenario, Saturn's performance drops drastically. This happens because clients now need to use the causality layer in order to migrate between datacenters. Since migration messages need to be propagated through the causality layer and wait for previous remote write operations to complete, the slow rate at which Saturn executes the remote write operations means that clients' migrate operations are en-queued behind a large amount of remote operations, thus needing a significant amount of time before they can be completed, leaving clients inactive for long periods of time. In contrast, in our solution, since we have more metadata and can execute remote operations concurrently, we avoid long remote operations queues hence, migration operations are much faster, avoiding clients that need to migrate from remaining stalled for long periods of time.

Taking into consideration the at operation's latency, and starting with the local only scenario, Figure 2b shows a big gap between the latency of write operations in the two systems. As explained previously, this is due to local write operations in Saturn not coordinating with the causality layer before being executed. Again, while this may make Saturn look better than our solution, it penalizes Saturn when both local and remote operations are executed.

Looking at the local and remote operation scenarios (Fig. 2e), we can see, just like in the throughput vs latency figures, a very different scenario. Writes have lower latency in Saturn since, as previously mentioned, they do not coordinate with the causality layer. The most interesting thing to note, however, is that the latency of migrate operations is particularly high in Saturn. This happens because it is the only operation that needs to go through the causality layer before the client receives a response, instead of just being executed in the local datacenter. The difference between our solution and Saturn is very significant (remember that the Y scale is logarithmic), leading to the overall latency to be higher in Saturn. As explained before, this happens due to the very slow execution of remote writes in Saturn, which results in very long queues of remote operations and migrations waiting to be executed in remote datacenters. This explains the performance of Saturn in Figure 2d.



(a) With only local operations



(b) With local and remote operations

Figure 3: Visibility times of each datastore configuration

5.3.3 Visibility Times

During the previous discussion, we mentioned the difference in the execution time of remote operations between our solution and Saturn. In Figure 3, we report these values. Results show the average visibility time (i.e how long it takes for a write operation to be executed in every datacenter) for each datastore configuration. Again, the Y axis in presented with a logarithmic scale. Table 3 reports the raw values for one of these experiences.

Since in both our solution and Saturn, visibility times are not affected by clients executing remote or local operations, results are similar in both figures. We can, however, observe a considerable difference between Saturn and C^3, which justifies our previous arguments that Saturn's visibility times are negatively affected due to limited concurrency in the execution of remote writes which, in turn, negatively affects the overall performance of the system.

5.4 Results Analysis

When compared with Cassandra, the overhead for providing stronger consistency guarantees (causal+ consistency instead of eventual consistency) is visible in the lower performance of our solution. However, we think that this cost is acceptable, especially when comparing with the Cassandra experiments that use the *EACH_QUORUM* consistency level, which can be seen as the Cassandra configuration that provides

#Clients	C-C3	C-SAT	E-EQ	E-LQ	E-Q
50	6,656	17,244	86	200	87
100	5,881	22,446	99	353	100
150	6,765	23,870	108	415	112
200	6,772	24,905	108	372	105
250	6,663	25,481	104	390	110
300	6,688	27,050	95	306	97
350	6,867	27,991	94	333	110

Table 3: Raw average visibility time (ms)

consistency guarantees that are closest to causal consistency (provided by our solution and Saturn). We also believe that there is still room for improvement in order to reduce this overhead, by optimizing our prototype implementation.

When compared with Saturn, our solution does not seem to be objectively better in every scenario. However, due to the migration of clients being unavoidable when operating under partial replication, our approach, by balancing the execution of local and remote operations, presents much better results. This leads us to conclude that having additional causality-tracking information between operations is a very important factor for systems providing causal consistency, as it enables more parallelism and faster execution of remote operations.

Due to the inefficiency of Saturn in executing remote operations concurrently in a datastore that supports sharding, it is also worth noting that, if we scale the datastore layer even further, by increasing the number of nodes and increasing the number of clients and operations being executed, it is predictable that the difference in results between our solution and Saturn will become even more noticeable.

6 Conclusion

In this paper, we presented an in-depth study of the challenges faced when attempting to create a solution capable of supporting causal consistency in a partial, geo-replicated setting. We exploited a novel approach to achieve causality, introduced by Saturn, which consists in separating the datastore layer from the causality tracking, allowing the causality layer to work with smaller pieces of data and hence low overhead.

As a result of this study, a novel causality tracking protocol was designed, created while taking in mind a partially, geo-replicated, and scalable underlying datastore, and requiring as few as possible modifications to the datastore system.

In addition to the design of this protocol, this paper also presents a concrete implementation of it. We implemented our protocol over a popular, eventual consistent datastore, Cassandra, describing the changes required to implement our solution on top of it. Experimental work was then conducted, using a real, partial, and geo-replicated test setting, to validate our solution, and compare it with the baseline datastore used and another causality tracking solution: Saturn.

The experimental results show that our protocol is capable of maintaining a good balance between the execution of local and remote operations, by using just enough metadata to allow concurrency in the execution of remote operations, which is essential to enable the datastore to scale through sharding.

References

[1] A. Silberstein et. al. Pnuts in flight: Web-scale data serving at yahoo. *IEEE Internet Computing*, 16(1):13–23, Jan 2012.

[2] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Eurosys*, pages 85–98. ACM, 2013.

[3] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. on Parallel and Dist. Sys.*, 28(1):141–155, 2017.

[4] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, pages 761–772. ACM, 2013.

[5] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[6] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Eurosys*, pages 111–126. ACM, 2017.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, pages 143–154. ACM, 2010.

[8] J. C. Corbett and et. al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.
[9] D. Akkoorath et. al. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, pages 405–414. IEEE, 2016.
[10] D. B. Terry et. al. Session guarantees for weakly consistent replicated data. In *ICPDIS*, pages 140–149, 1994.
[11] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *SOCC*, pages 11:1–11:14. ACM, 2013.
[12] F. Freitas, J. Leitão, N. Preguiça, and R. Rodrigues. Fine-Grained Consistency Upgrades for Online Services. In *SRDS*, pages 1–10, 2017.
[13] Giuseppe DeCandia et. al. Dynamo: amazon's highly available key-value store. *SIGOPS Op. Sys. Rev.*, 41(6):205–220, 2007.
[14] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Trade-offs in replicated systems. *IEEE Data Eng. Bulletin*, 39:14–26, 2016.
[15] C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *USENIX ATC 17*, pages 83–95. USENIX Ass., 2017.
[16] H. Mahmoud et. al. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9).
[17] H. Moniz et. al. Blotter: Low latency transactions for geo-replicated storage. In *WWW*, pages 263–272. Int. WWW Conf. Steer. Comm., 2017.
[18] J. Du et. al. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proc of SOCC*, pages 1–13. ACM, 2014.
[19] R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.
[20] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *PODC*, pages 43–57. ACM, 1990.
[21] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Op. Sys. Rev.*, 44(2):35–40, 2010.
[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7).
[23] M. Bravo et. al. On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull.*, 38(1):18–31, 2015.
[24] P. Mahajan, L. Alvisi, M. Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin TR*, 11, 2011.
[25] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1).
[26] W. Lloyd et. al. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *SOSP*, pages 401–416. ACM, 2011.
[27] W. Lloyd et. al. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, pages 313–328, 2013.