MULTI-VALUE DISTRIBUTED KEY-VALUE STORES

RICARDO JORGE TOMÉ GONÇALVES

The MAP-i Doctoral Program in Computer Science of the Universities of Minho, Aveiro and Porto

December 2017

Ricardo Jorge Tomé Gonçalves: *Multi-Value Distributed Key-Value Stores,*
The MAP-i Doctoral Program in Computer Science of the Universities
of Minho, Aveiro and Porto, © December 2017

People think that computer science is the art of geniuses,
but the actual reality is the opposite,
just many people doing things that build on each other,
like a wall of mini stones.

— Donald Knuth

ACKNOWLEDGEMENTS

I would like to express my special gratitude to Professor Paulo Sérgio Almeida, for being my advisor and supporting my work with great advise and encouragement. Also a special thanks to Professors Carlos Baquero and Vitor Fonte, both heavily involved in this work. To all of them, thank you for the continuous support and encouragement throughout this work and for the counseling provided. I learned a lot. Without their guidance and dedication this thesis would not have been possible.

Thanks to my all of my colleagues and friends at the HASLab laboratory, for the great work environment created. Additionally, a special thanks to my Ana and my family, for all the love and support given, and for always having confidence in me.

Finally, this work was funded by National Funds through the FCT — *Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) , with the research grant SFRH/BD/86735/2012.

ABSTRACT

Many large scale distributed data stores rely on optimistic replication to scale and remain highly available in the face of network partitions. Managing data without strong coordination results in eventually consistent data stores that allow for concurrent data updates. To allow writing applications in the absence of linearizability or transactions, the seminal Dynamo data store proposed a multi-value API in which a get returns the set of concurrent written values. In this scenario, it is important to be able to accurately and efficiently identify updates executed concurrently. Logical clocks are often used to track data causality, necessary to distinguish concurrent from causally related writes on the same key. However, in traditional mechanisms there is a non-negligible metadata overhead per key, which also keeps growing with time, proportional to the node churn rate. Another challenge is deleting keys while respecting causality: while the values can be deleted, per-key metadata cannot be permanently removed in current data stores.

These systems often use anti-entropy mechanisms (like *Merkle Trees*) to detect and repair divergent data versions across nodes. However, in practice hash-based data structures are not suitable to a store using consistent hashing and create too many false positives.

Also, highly available systems usually provide eventual consistency, which is the weakest form of consistency. This results in a programming model difficult to use and to reason about. It has been proved that causal consistency is the strongest consistency model achievable if we want highly available services. It provides better programming semantics such as sessions guarantees. However, classical causal consistency is a memory model that that is problematic for concurrent updates, in the absence of concurrency control primitives. Used in eventually consistent data stores, it leads to arbitrating between concurrent updates which leads to data loss.

We propose three novel techniques in this thesis. The first is Dotted Version Vectors: a solution that combines a new logical clock mechanism and a request handling workflow that together support the traditional Dynamo key-value store API while capturing causality in an accurate and scalable way, avoiding false conflicts. It maintains concise information per version, linear only on the number of replicas, and includes a container data structure that allows sets of concurrent versions to be merged efficiently, with time complexity linear on the number of replicas plus versions.

The second is DottedDB: a Dynamo-like key-value store, which uses a novel node-wide logical clock framework, overcoming three fundamental limitations of the state of the art: (1) minimize the metadata per key necessary to track causality, avoiding its growth even in the face of node churn; (2) correctly and durably delete keys, with no need for tombstones; (3) offer a lightweight anti-entropy mechanism to converge replicated data, avoiding the need for Merkle Trees.

The third and final contribution is Causal Multi-Value Consistency: a novel consistency model that respects the causality of client operations while properly supporting concurrent updates without arbitration, by having the same Dynamo-like multi-value nature. In addition, we extend this model to provide the same semantics with read and write transactions. For both models, we define an efficient implementation on top of a distributed key-value store.

RESUMO

Várias bases de dados de larga escala usam técnicas de replicação otimista para escalar e permanecer altamente disponíveis face a falhas e partições na rede. Gerir os dados sem coordenação forte entre os nós do servidor e o cliente resulta em bases de dados "inevitavelmente coerentes" que permitem escritas de dados concorrentes. Para permitir que aplicações escrevam na base de dados na ausência de transações e mecanismos de coerência forte, a influente base de dados *Dynamo* propôs uma interface multi-valor, que permite a uma leitura devolver um conjunto de valores escritos concorrentemente para a mesma chave. Neste cenário, é importante identificar com exatidão e eficiência quais as escritas efetuadas numa chave de forma potencialmente concorrente. Relógios lógicos são normalmente usados para gerir a causalidade das chaves, de forma a detetar escritas causalmente concorrentes na mesma chave. No entanto, mecanismos tradicionais adicionam metadados cujo tamanho cresce proporcionalmente com a entrada e saída de nós no servidor. Outro desafio é a remoção de chaves do sistema, respeitando a causalidade e ao mesmo tempo não deixando metadados permanentes no servidor.

Estes sistemas de dados utilizam também mecanismos de anti-entropia (tais como *Merkle Trees*) para detetar e reparar dados replicados em diferentes nós que divirjam. No entanto, na prática estas estruturas de dados baseadas em *hashes* não são adequados para sistemas que usem *hashing consistente* para a partição de dados e resultam em muitos falsos positivos.

Outro aspeto destes sistemas é o facto de normalmente apenas suportarem coerência inevitável, que é a garantia mais fraca em termos de coerência de dados. Isto resulta num modelo de programação difícil de usar e compreender. Foi provado que coerência causal é a forma mais forte de coerência de dados que se consegue fornecer, de forma

a que se consiga também ser altamente disponível face a falhas. Este modelo fornece uma semântica mais interessante ao cliente do sistema, nomeadamente as garantias de sessão. No entanto, a coerência causal tradicional é definida sobre um modelo de memória não apropriado para escritas concorrentes não controladas. Isto leva a que se arbitre um vencedor quando escritas acontecem concorrentemente, levando a perda de dados.

Propomos nesta tese três novas técnicas. A primeira chama-se *Dotted Version Vectors*: uma solução que combina um novo mecanismo de relógios lógicos com uma interação entre o cliente e o servidor, que permitem fornecer uma interface multi-valor ao cliente similar ao *Dynamo* de forma eficiente e escalável, sem falsos conflitos. O novo relógio lógico mantém informação precisa por versão de uma chave, de tamanho linear no número de réplicas da chave no sistema. Permite também que versão diferentes sejam corretamente e eficientemente reunidas.

A segunda contribuição chama-se *DottedDB*: uma base de dados similar ao *Dynamo*, mas que implementa um novo mecanismo de relógios lógicos ao nível dos nós, que resolve três limitações fundamentais do estado da arte: (1) minimiza os metadados necessários manter por chave para gerir a causalidade, evitando o seu crescimento com a entrada e saída de nós; (2) permite remover chaves de forma permanente, sem a necessidade de manter metadados indefinidamente no servidor; (3) um novo protocolo de anti-entropia para reparar dados replicados, de modo a que todas as réplicas na base de dados convirjam, sem que seja necessário operações dispendiosas como as usadas com *Merkle Trees*.

A terceira e última contribuição é Coerência Causal Multi-Valor: um novo modelo de coerência de dados que respeita a causalidade das operações efetuadas pelos clientes e que também suporta operações concorrentes, sem que seja necessário arbitrar um vencedor entre as escritas, seguindo o espírito da interface multi-valor do *Dynamo*. Adicionalmente, estendemos este modelo para fornecer transações de escritas ou leituras, respeitando a mesma semântica da causalidade. Para ambos os modelos, definimos uma implementação eficiente em cima de uma base de dados distribuída.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

CAP Consistency-Availability-Partition Tolerance

CH Causal Histories

VC Vector Clocks

VV Version Vector

DVV Dotted Version Vectors

DVVS Dotted Version Vector Sets

CRDT Conflict-free Replicated Data Types

AE Anti-Entropy

DC Data-Center

NDC Node-wide Dot-based Clocks

NDC-NF Node-wide Dot-based Clocks Without Filling

CM Causal Memory

CC Causal Consistency

CMV Causal Multi-Value

CMVM Causal Multi-Value Memory

TxCMV Transactional Causal Multi-Value

TxCMVM Transactional Causal Multi-Value Memory

1

INTRODUCTION

Internet-scale distributed systems are typically put together as a mix of different applications and sub-systems [59], often combining different trade-offs with respect to choices of consistency and availability in the face of partitions [12, 20].

Dynamo [15] popularized the distributed data store with high availability and weaker consistency. The shopping cart was one of the use cases presented by the Amazon team to motivate the need for high-availability systems. If a user chose an item that was said to be in stock and later in the checkout it was not available, Amazon would simply apologize to the user. This was preferred to the alternative, where demanding strong consistency would cause more unresponsiveness or downtime to the Amazon site, which would result in less revenue.

Nowadays, many distributed data stores [15, 29, 33] depart from the stronger consistency guarantees that can be provided in relational databases; instead, they offer scalable solutions and choose to stay available rather than consistent, accepting the impact of data divergence when partitions occur. Moreover, they allow low latency responses even when nodes are geographically spread. These properties were a strong motivation in the industry to migrate some applications that had lower consistency requirements compounded by higher availability and timing concerns, while leaving others in classic relational solutions or enlisting services that provided stronger coordination [25].

These systems allow writing concurrently on different nodes and reading possibly stale data. This avoids the need for global coordination to totally order writes, but possibly creates data divergence. To

deal with conflicting versions for the same key, generated by concurrent writes, they can either use the *last-writer-wins* rule [26], which only keeps the "last" version (according to a wall-clock timestamp for example) and discard the other versions, therefore losing data, or we can properly track each key causal history with logical clocks [34], which track a partial order on all writes for a given key, to detect concurrent writes.

Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [10, 34, 46, 52, 53]. In particular, for data storage systems, Version Vectors [46] enable the system to compare any pair of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete. However, they lack the ability to accurately represent concurrent values when used with server ids, or are not scalable when used with client ids.

However, even with accurate tracking of write conflicts, nodes still can diverge due to message loss or temporary network partitions. Anti-entropy protocols are often used as a background process that detects and repairs data inconsistencies. Merkle Trees [43] are the most used structure to detect data differences in distributed databases (e.g., Cassandra [33]and Riak [29]). However, as we will show here, Merkle Trees are not suitable to be used together with the normally used consistent hashing, and lead to weak anti-entropy performance.

Although many distributed data stores only offer the weakest form of consistency — eventual consistency —, causal consistency [1] is one of the stronger consistency models that still allows for high availability [6, 39]. It is an attractive model for distributed data stores, because it guarantees that a data item only becomes visible to the client after satisfying all of its dependencies. This eliminates many anomalies found in systems that only implement eventual consistency, the weakest consistency model used in practice. Causal consistency also allows the client to write in any node that replicates the data item and avoids having serialization bottlenecks in the system, lowering the latency seen in operations when compared to stronger models.

However, in classic causal memory, a read returns a single value. For shared-memory concurrent programming in multi-cores, even when weak memory models are offered, synchronization primitives can be used to, e.g., prevent data-races or enforce mutual-exclusion. In a distributed data store, without providing programs with synchronization primitives, having the classic single-value memory API will result in lost updates, when several processes concurrently perform a read-compute-write update to the same location. This happens in most systems that offer causal consistency, such as COPS [36] and Eiger [37].

1.1 PROBLEM STATEMENT AND OBJECTIVES

Multi-Value from Write-Write Conflicts

Distributed data stores that choose to sacrifice data consistency for greater availability have to support clients that concurrently write to the same object. There are two main approaches to detect these *write-write* conflicts: wall-clock timestamps and logical clocks.

Timestamps are easier to implement, but rely on globally synchronized physical clocks. Even with perfect physical clocks, which cannot be achieved in practice [44], they would still fail to capture the *causality* [34] between updates. Two clients can read the same object, with a given key, and each write an updated version to some server node. A totally ordered timestamp cannot express this concurrent update pattern. With timestamps, a last-writer-wins policy is typically used, causing an arbitrary loss of all but one of the concurrent updates.

Logical clocks capture causality between object versions. We use the term *logical clock* to denote any non-physical clock (not only Lamport scalar clocks [34], but also, e.g., vector clocks [19, 42]). Version Vectors [46] would detect the update conflict above, since updates originated from the same object and diverged independently. Logical clocks can either have server-based ids or client-based ids. The latter are prohibitive (with many clients), while node-ids have a space complexity linear with the number of active nodes over an object's lifetime. There-

fore, logical clocks may induce significant metadata costs and require further optimization in order to compete with the space efficiency of wall-clock timestamps.

Logical Clocks and Node Churn

When a node is retired or crashes permanently, its node-id remains present in logical clocks for object replicas in other live-nodes. It is also propagated to replicas in the new replacement nodes, which themselves will introduce a new node-id. Over time, the size of version-vector-like logical clocks will increase with the total number of nodes ever used. In Dynamo and early versions of Riak there is a limit to the size of the vector, that when reached induces a removal of entries via a LRU policy. However, removing entries is not safe in general and can lead to false conflicts: wrongly identifying two versions as causally concurrent, when in fact one *happened-before* [34] the other, becoming obsolete.

Distributed Deletes

Given a delete request, completely removing an object and corresponding logical clock information from storage is normally not possible without losing causality information, which may lead to that object resurfacing via delayed replication messages or synchronization with outdated nodes. In current schemes, the payload of the key can be removed, but the key must continue to map to the current logical clock paired with a tombstone, to ensure that causality is respected across replica nodes. The extra metadata required per deleted key results in space consumption which is linear with the number of deleted keys, leading to significant waste over time. It would be desirable to have an alternative that does not require metadata for deleted keys. This is important for small payloads where the relative cost of keeping tombstones is higher.

Efficient Anti-Entropy

The network is unreliable [7] and often replication messages are lost or nodes are simply partitioned and cannot communicate. This precludes write operations from disseminating the written data (and metadata) to all replicas. Over time, for any given pair of nodes, the replicas corresponding to the subset of keys in common diverge and must be repaired towards convergence. This is typically done periodically, in bulk, by *Anti-Entropy* protocols [16] running as a background task.

The most used anti-entropy protocol in distributed databases involves hashing objects into a Merkle Tree [43] and then comparing trees to detect differences. This approach makes a tradeoff between tree size (branching factor and tree depth) and false positives in objects that must be repaired. Communication costs can be high in both cases: with a large tree, we exchange a lot of metadata to learn which objects must be repaired; with a small tree, we exchange a large amount of key-object hashes, even with a few updates. The impact is so significant that anti-entropy is turned off by default in some systems, with convergence limited to *read-repair*: relevant replicas are updated when nodes detect inconsistencies while serving a client read.

Multi-Value Causal Consistency

Another common complaint about these systems is their lack of guarantees from the client's perspective. This is something that certainly harms the adoption of fully distributed data stores and we think that something like session guarantees [56] makes a big difference in what the client can expect from the data store, without going to a fully consistent design. However, there are no consistency models that preserve the high-availability nature of distributed data stores and properly support concurrent writes, as mentioned before.

Defining a protocol implementing causal consistency that is efficient and scalable with its use of dependencies, is a well known research challenge, with multiple takes found in the literature [5, 17, 18, 36, 37, 55,

62]. Some systems require data to fit in a single machine, while others ignore the problem of concurrent operations executed by multiples clients. In general they all use a single-value memory model.

In this work, our objective is to develop a causal consistency model, mechanisms and algorithms exposing a multi-value API (in the spirit of Dynamo), to provide highly available, scalable and distributed data stores that allow concurrent updates without losing data.

1.2 CONTRIBUTIONS AND RESULTS

Over the course of this thesis, we make several contributions to the state-of-the-art of distributed data stores. These contributions can be divided in three parts: an improved logical clock for causality tracking; a new framework for distributed data stores with an anti-entropy protocol, distributed deletes, and scalable metadata for causality tracking; a new causal consistency model with support for concurrent values and an extension of that model for read and write transactions.

Dotted Version Vector Sets (DVVS) are a new logical clock that supports encoding multiple concurrent writes originated from clients, while using server-based identifiers. This mechanism differs from previous mechanisms because it enables clients to independently resolve a subset of the concurrent writes. The result is a logical clock that has better space and time complexity that previous state-of-the-art mechanisms.

Node-wide Dotted-based Clocks (NDC) is a novel framework for distributed data stores that uses per-node logical clocks to uniquely tag every new write to the system. The node logical clock represents exactly what writes are present at that node, or were overwritten by current writes. With this information, it provides a lightweight anti-entropy protocol for detecting and repairing stale data. Another benefit of having the node logical clock is smaller and scalable causality metadata for objects, taking inspiration from DVVS. Finally, it enables distributed deletes to execute without leaving permanent metadata behind (this metadata is often called *tombstones*). We provide two competing versions of NDC, one that removes and restores metadata from objects and

another that only removes metadata from objects. The former has better support for multiple node failures, while the latter is more efficient and simpler. Another contribution from this work is DottedDB, our distributed data store that implements both versions of NDC.

Lastly, we propose and formalize a new consistency model: Causal Multi-Value Memory (CMVM). Causal consistency is often said to be the strongest consistency model that supports a highly-available system. CMVM provides the same semantics as causal consistency but supports concurrent write operations while avoiding the need to arbitrate which write wins in the presence of conflicts, by allowing a read to return a set of values. We also provide a reference implementation for CMVM in a distributed data store, building on NDC as a foundation. In addition, we extend CMVM to support read-only and write-only transactions, along with an appropriate reference implementation.

PUBLICATIONS Some of the work described in this thesis was published in several international conferences:

- Scalable and Accurate Causality Tracking for Eventually Consistent Stores. Paulo Sérgio Almeida, Ricardo Gonçalves, Carlos Baquero Moreno, Vitor Fonte. IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2014.
- Concise Server-Wide Causality Management for Eventually Consistent Data Stores. Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero Moreno, Vitor Fonte. IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2015.
- DottedDB: Anti-Entropy without Merkle Trees, Deletes without Tombstones. Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero Moreno, Vitor Fonte. IEEE International Symposium on Reliable Distributed Systems (SRDS), 2017.

1.3 OUTLINE

The rest of this thesis is organized as follows:

- Chapter 2 gives a brief presentation on the necessary background and some of the state of the art literature relevant to the rest of the thesis;
- Chapter 3 presents Dotted Version Vector Sets, a strictly-better alternative to version vectors for managing data versions in distributed key-value stores;
- Chapter 4 presents Node-wide Dot-based Clocks, a framework for distributed key-value stores that provides a lightweight anti-entropy protocol, support for distributed deletes and space-efficient version tracking;
- Chapter 5 presents Causal Multi-value Consistency, a new consistency model that supports causality while providing a multi-value API. It also defines an reference implementation for a distributed key-value store supporting causal multi-value consistency;
- Chapter 6 presents and extension to the causal multi-value model defined in the previous chapter, introducing read-only and write-only transactions. We define the extended memory model and its implementation in a distributed key-value store;
- Chapter 7 is the conclusion of the thesis with some remarks about future work directions.

2

BACKGROUND

This chapter focuses on providing the necessary background for work described in the subsequent chapters. Central to this thesis work is the notion of data consistency, which ranges from the strong consistency models like linearizability to weak consistency models such as eventual consistency. Given our focus on weaker consistency models for higher availability, scalability and fault-tolerance, we also explore state of the art solutions to data synchronization for replicated data and data versioning with logical clocks. We also discuss the typical architecture for distributed weakly consistent key-value stores.

2.1 CONSISTENCY

If we look at two replicated nodes at any given time, they will probably be in different states since writes are applied first in one node, and replicated to others next. This delay is inevitable no matter the underlying replication mechanism (e.g.: single-master or leaderless replication) and creates windows of opportunity for clients to observe inconsistent states.

Furthermore, even with zero latency replication, failures can halt or delay communication between replica nodes, which forces the system to choose between staying consistent by possibly rejecting client requests, or accept all requests, but accepting that data may diverge to inconsistent states. This choice was popularized by the CAP theorem [12, 20], with the catch-phrase: *given Consistency, Availability and Partition-Tolerance, chose 2 of the 3*.

Although the theorem implies that are three combinations, a distributed system cannot simply not choose partition-tolerance as they are part of failures that will happen. Therefore, the real design choice is between CP systems (consistent) and AP systems (available). Also, *consistency* here only means Linearizability, where in fact the distributed systems literature offers several consistency models that offer a spectrum of guarantees and their trade-offs.

We will now briefly describe some of the relevant consistency models.

2.1.1 *Linearizability*

Linearizability [24] (also called *Atomic Consistency*, *Immediate Consistency*, *External Consistency* or *Strong consistency*) is the strongest form of consistency which, informally explained, gives the clients the illusion of accessing objects with a single variable with atomic operations.

It provides the abstraction of a global total order for all operations, however, clashes with physical reality, imposing delays (blocking operations) on each participant, hindering spatial scalability.

There are costs with linearizability: availability in face of failures (like network partitions) and performance. The latter can be also seen even in tightly coupled systems such as multi-processors: for performance reasons, variables across threads in different CPU cores are not guaranteed linearizability, since it would slow down the CPU [45].

2.1.2 *PRAM*

PRAM [35] (Pipelined RAM) memory model only ensures that the writes of each process are seen in program order by other processes (but allowing writes from different processes to be seen in different orders at different processes). This is simple to achieve but overly weak, as it admits many executions which are confusing to programmers.

2.1.3 *Causal Consistency*

Causal memory [1] only ensures that *causally related* operations are seen correctly ordered by every process (allowing concurrent operations to be seen in different orders). Enforcing causal relations is an interesting point in the consistency spectrum: it is the strongest consistency that can be achieved without compromising availability [6, 39], while still performing better than linearizability (since it is less sensitive to network delays). More generally, it does not impose delays proportional to the system spatial span, as causality propagation is limited to the speed of light, and distant "space-like" events cannot be causally related, and do not need to be ordered. This, together with the intuitive appeal to programmers, has made causal consistency increasingly pursued as the consistency model to offer in distributed key-value stores which aim for availability.

CAUSAL+ CONSISTENCY Causal+ Consistency was defined by Loyd et al. in [36] and it is an extension of traditional causal consistency with an additional rule that states that objects must deterministically converge at every replica node. Although it is expected that nodes should communicate and exchange data in order to converge their replicated data, the original causal memory definition did not explicitly state that different replicas should converge, allowing different processes to see different values forever. In Causal+, concurrent writes to the same key must be eventually resolved to a deterministic result.

2.1.4 *Eventual Consistency*

Eventual Consistency [60] is what most databases offer as the weakest consistency level. It states that if writes to the database stop, eventually (i.e., after some unspecified time) the replicated state will be consistent, returning the same value to every process. Implicitly, there is the assumption that data aims to be stored, ruling out vacuous implementations which returned the same value forever.

This consistency model can be thought simply as convergence guarantee, without any time guarantee.

2.2 DATA SYNCHRONIZATION

The normal client request handling workflow may leave several replicas with different versions, either because of message loss or network partitions. There must be a further mechanism by which the system can self-heal and achieve consistency, even if no more client requests arrive: outdated objects should be replaced by newer ones, new objects should be present in all relevant nodes and deletes should remove the corresponding replicas in all relevant nodes. Such a mechanism is called an anti-entropy protocol [16], typically being run periodically between pairs of nodes. The two most common data structures used to repair replicated data are *Bloom Filters* [11] and *Merkle Trees* [43]. Most Dynamo-like systems today use a protocol based on Merkle trees to compute the differences between nodes, since bloom filters are not as efficient for large data-sets [30]

2.2.1 *Bloom Filters*

Bloom filter are well known for their useful semantics, low overhead and probabilistic answers. There are many types of bloom filters. In general, all of them allow an item to be tested as belonging to a set, which is represented by the bloom filter. The classical bloom filter may give false positives but never false negatives, i.e., for a given item, a bloom filter can answer that it is *probably* in the set, or *not* in the set.

2.2.2 *Merkle Trees*

A Merkle tree is tree data structure where: each inner node stores the hash of its children hashes; the leaf nodes store a list of key-hash pairs and the hash of that list. Since peers may not replicate the same set of keys, due to partial replication, each server node maintains one Merkle

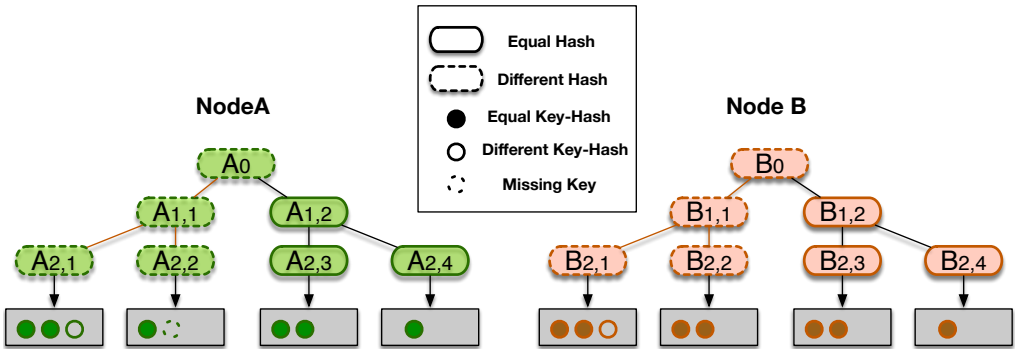


Figure 2: Two Merkle trees that represent the state of the objects replicated by nodes A (left) and B (right). In a level-by-level exchange of the Merkle tree, these two nodes end up exchanging the hashes: $\{A_0, B_0, A_{1,1}, B_{1,1}, A_{1,2}, B_{1,2}, A_{2,1}, B_{2,1}, A_{2,2}, B_{2,2}\}$. Node A misses an object on leaf $A_{2,2}$, present in the corresponding B's leaf $B_{2,2}$. Node B has a conflicting object version on leaf $B_{2,1}$, when comparing to A's leaf $A_{2,1}$.

tree per replica group: a set of peers that replicates a common subset of keys. Each new object is hashed and inserted in the appropriate leaf node, according to its key-hash. The hash of the leaf node is then updated, as are all parent nodes until the root.

To compare the state of two nodes, they exchange the corresponding Merkle trees by levels, only descending further down the tree if the corresponding hashes are different. If two corresponding leaf nodes have different hashes, then there are objects which must be repaired. The list of (key,hash) pairs is then exchanged and the final list of objects in need of repair is computed.

Figure 2 shows an example of two Merkle trees representing the same set of objects replicated in two different nodes. Node A has an object in leaf $A_{2,1}$ that has a different hash on node B, while Node B has an object in leaf $B_{2,2}$ that is missing from Node A. Both nodes start by comparing the root node and confirm that the hash is different, so they exchange and compare the next level of the tree. On the second level, only one node has a hash mismatch, and the protocol continues only for its children.

ACCURACY VERSUS METADATA There is a fundamental trade-off in Merkle trees, between the tree size and the accuracy of the detection of

outdated keys. The size of the tree, given by the number of children per node (branching factor) and number of levels (height of the tree) should not be considered in absolute terms but relative to the number of keys that are stored. An appropriate metric to evaluate accuracy-versus-size is the number of keys per leaf node.

The optimal setting for accuracy would be a key/leaf ratio around 1, where ideally all leaf nodes would be used, storing the hash of only one object, and therefore being always accurate. However, this is impractical for large datasets, as it would imply a large tree size, even with a dynamically sized tree [30].

2.2.3 *Read Repair*

Although Read Repair does not repair data in bulk as do Merkle Trees or Bloom Filters, it is a simple mechanism of detecting inconsistent data when a client performs a read that asks more that one replica node for its content, as usual in quorum-based systems. Since the coordinator of the read request receives several copies for the same key, it can compare them and issue update requests to outdated replicas.

The issue with read repair is that rarely read objects may be inconsistent for a long time. However, this is a passive mechanism that can be used with other proactive mechanisms that run in the background.

2.3 LOGICAL CLOCKS FOR CAUSALITY TRACKING

The role of causality in distributed systems was introduced by Lamport [34], establishing the foundation for the subsequent mechanisms and theory [10, 13, 34, 46, 52, 53].

2.3.1 *Single-Object Logical Clocks*

Replicated objects may diverge when multiple clients concurrently update the same key. When a node receives an object to store and it already

has another version, it must decide how it will choose between them or if it will keep both.

2.3.1.1 *Physical Timestamps*

Some systems like Cassandra [33] tag each version of an object with a physical timestamp and use a last-writer-wins (`LWW`) policy. With timestamps, each node keeps the version with the newest timestamp. Although extremely simple to implement and use, there are a couple of problems with this approach: first, we have to trust the entity issuing the timestamps; clients can maliciously lie about their timestamp to supersede other versions, machine clocks can drift apart, the NTP server could be offline, etc. Second, even if the timestamps were perfect, it still would not capture the true nature of the concurrent client updates, since by definition only one would survive, and the other would be lost (without the client noticing).

2.3.1.2 *Per-Object Logical Clocks*

Others systems use the notion of causality [34] to tag versions with some logical clock, which does not rely on real time, and allows detecting when two updates are concurrent. This enables preserving both versions to be reconciled later by a deterministic algorithm or by the client.

A drawback is that each per-object logical clock has a size linear with the replication factor (ignoring logarithmic factors). While this may not seem problematic with the usual small replication factors (e.g., 3 replicas), node churn is a natural occurrence in distributed systems, with new nodes replacing failed nodes over time. This in turn will pollute the logical clock with more and more entries over time. The normal approach to overcome this problem is to remove "old" metadata, but this breaks causality and introduces false concurrency into the system.

CAUSAL HISTORIES Causal Histories [53] (CH) are simply described by sets of unique write identifiers. These identifiers can be generated with a unique identifier and a monotonic counter. Let id_n be the no-

tation for the n^{th} event of the entity represented by *id*. The crucial point is that identifiers have to be globally unique to correctly represent causality. The partial order of causality can be precisely tracked by comparing these sets under set inclusion. Two CH are concurrent if neither includes the other: $A \parallel B$ iff $A \not\subseteq B$ and $B \not\subseteq A$. CH correctly track causality relations, but have a major drawback: they grow linearly with the number of writes.

VERSION VECTORS Version Vectors (VV) are an efficient representation of CH provided that the CH has no gaps in each *id*'s event sequence. A VV is a mapping from identifiers to counters, and can be written as a set of pairs $(id, counter)$; each pair represents a set of CH events for that *id*: $\{id_n \mid 0 < n \leqslant counter\}$. In terms of partial order, $A \leqslant B$ iff $\forall (i, c_a) \in A \cdot \exists (i, c_b) \in B \cdot c_a \leqslant c_b$. Again, $A \parallel B$ iff $A \not\leqslant B$ and $B \not\leqslant A$. Whether client or server identifiers are used in VV has major consequences, as we'll see next.

VARIABILITY IN THE NUMBER OF ENTITIES The basic vector based mechanisms can be generalized to deal with a variable number of nodes or replicas. The common strategy is to map identifiers to counters and handle dynamism in the set of identifiers. Additions depend on the generation of unique identifiers. Removals can require communication with several other servers [21], or to a single server [3, 47]. Interval Tree Clocks [3] are specifically designed to track causality in dynamic scenarios but scalability depends on recovery of identifiers.

The Dependency Sequences [48] mechanism assumes a scenario where dynamic, weakly-connected sets of entities (mobile hosts) communicate through designated proxy entities chosen from a stable, well-connected (mobile service stations) network. The mechanism maintains information about the causal predecessors of each event.

CONDITIONAL WRITES Some systems just detect the concurrent PUT operations from different clients and reject the update (e.g. version control systems such as CVS and subversion) or keep the updates but do not allow further accesses until the conflict is solved (e.g. original ver-

sion of Coda [28]). In these cases, using version vectors (VV) with one entry per server is sufficient, as a concurrent version can be detected in a server if the VV obtained by the client when reading the data is different from the VV when it executes the PUT. However, these solutions sacrifice write availability which is a key "feature" of modern geo-replicated databases.

COMPACTING THE REPRESENTATION In general, using a format that is more compact than the set of independent entities that can register concurrency, leads to lossy representation of causality [13]. Plausible clocks [58] condense event counting from multiple replicas over the same vector entry, resulting in false concurrency. The resulting order does not contradict causality but some concurrent events are perceived as causally related, because counters are effectively shared between processes. In fact, the previously mentioned Lamport clocks [34] are a notable example of plausible clocks.

Several approaches for removing entries that are not necessary have been proposed, some being safe but requiring running consensus (e.g. Roam [51]), and others fast but unsafe (e.g. Dynamo [15]) potentially leading to causality errors.

EXTENSIONS AND ADDED EXPRESSIVENESS In Depot [38], the VV associated with each update only includes the entries that have changed since the previous update in the same node. However, each node still needs to maintain VV that include entries for all clients and servers; in a similar scenario, the same approach could be used as a complement to our solution. Other systems explore the fact that they manage a large number of objects to maintain less information for each object. WinFS [41] maintains a base VV for all objects is the file system, and for each object it maintains only the difference for the base in a concise VV.

Cimbiosys [50] uses the same technique in a peer-to-peer system. These systems, as they maintain only one entry per server, cannot generate two VV for tagging concurrent updates submitted to the same server from different clients, as discussed in Section 3.2 with VV_{server}.

WinFS includes a mechanism to deal with disrupted synchronizations that allows encoding non sequential causal histories by registering exceptions to the events registered in VV; e.g. $\{a_1, a_2, b_1, c_1, c_2, c_4, c_7\}$ could be represented by $\{(a, 2), (b, 1), (c, 7)\}$ plus exceptions $\{c_3, c_5, c_6\}$.

Wang et. al. [61] have proposed a variant of VV with $O(1)$ comparison time (like our own DVV), but the VV entries must be kept ordered, which prevents constant time for other operations.

2.3.2 *Multi-Object Logical Clocks*

While the basic usage of logical clocks involves treating each object independently, to overcome the per-object metadata size overhead, a powerful idea is to factor out knowledge common to the whole node into a *Node Logical Clock* to supplement each object logical clock, making the per-object clock smaller.

Ladin et al. [31] developed Lazy Replication with node logical clocks and a Lamport clock [34] per write, but the metadata compaction depends on loosely-synchronized clocks and the availability of client replicas.

Bayou [57] attaches writes with a Lamport clock and stores them in three different logs, each with its own logical clock: the tentative writes, the committed writes and the undo writes. This works because it totally orders writes with a primary server, in order to store only the maximum counter per replica.

Eiger [37] focus is on causal consistency [1] by using one Lamport clock per node, to issue globally unique ids to updates, but it does not support concurrent versions, nor does it address anti-entropy repair.

Concise Version Vector (CVV) [41] uses a node clock to repair replicas in a distributed file system, but it does not address how to deal with distributed deletes, does not provide a detailed algorithm to identify which keys are missing, their object logical clock is not bounded by the replication factor, and it lacks support for concurrent versions.

Vector Sets [40] improve on CVV by placing an upper-bound on the size of object logical clocks, dividing the objects in sets that can be rep-

resented by a single compact version vector, instead of a single node logical clock.

Cimbiosys [50] also builds upon CVVs to build a peer-to-peer partial replication platform, but also fails to support concurrent values, and its anti-entropy is inefficient since it sends all potential missing keys to a replica.

2.4 WEAKLY-CONSISTENT DATA STORES

Amazon's Dynamo system [15] was an important influence to a new generation of databases, such as Cassandra [32, 33] and Riak [29], focusing on partition tolerance, write availability and eventual consistency. The underlying rationale to these systems stems from the observation that when faced with the three concurrent goals of *consistency*, *availability* and *partition-tolerance* only two of those can be achievable in the same system [12, 20]. Facing geo-replication operation environments where partitions cannot be ruled out, consistency requirements are inevitably relaxed in order to achieve high availability.

These systems follow a design where the data store is always writable: replicas of the same data item are allowed to temporarily diverge and to be repaired later on. A simple repair approach followed in Cassandra, is to use wall-clock timestamps to know which concurrent updates should prevail. This last writer wins (`LWW`) policy may lead to lost updates. An approach which avoids this, must be able to represent and maintain causally concurrent updates until they can be reconciled.

Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [10, 34, 46, 52, 53]. In particular, for data storage systems, version vectors (`VV`) [46] enable the system to compare any pair of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete. However, `VV` lack the ability to accurately represent concurrent values when used with server ids, or are not scalable when used with client ids.

Weakly-consistent data stores provide consistency guarantees weaker than linearizability, but remain available in the presence of arbitrary network partitions.

2.4.1 *Architecture*

The data store is replicated across R nodes and its uses a *leaderless* replications strategy, which means that each node is independent from the others and it can serve at any time a read or write request to a key that it replicates. Nodes communicate via message passing and messages can be delayed, lost and reordered. The key-range is also partitioned in several ranges for scalability. This is typically done with consistent hashing, where nodes are placed in a position in an abstract ring. Keys are also hashed to a position in the ring and the first R nodes after that position, are the replica nodes for that key.

PARTIAL REPLICATION WITH PARTIAL OVERLAPPING Distributed key-value stores may serve many thousands of clients, and need to store large data-sets spread over many (e.g., hundreds) nodes, each data item stored in several (e.g., three) replicas. This means that each node has only some part of all data (partial replication). Also, keys are placed usually by consistent hashing, and the replication scheme (e.g., in Riak [29]) may result in no two nodes storing the same set of keys (partial overlapping). This combination of partial replication and partial overlapping is challenging for providing causal consistency (i.e., cross-object causality), contrary to a an implementation assuming full replication (such as in the original causal memory paper [1]).

ANTI-ENTROPY Since weakly-consistent systems allow nodes to operate without strong coordination with other nodes, data can become out-of-sync given failures (e.g. message loss, network partitions, node crashes, etc.). Given this reality, the system must provide a mechanism to repair data that is out-of-sync between nodes.

Read repair is a technique where the node coordinating a read request detects if nodes responded with outdated data and sends the newer data to it. That helps, but is not enough since it only repairs nodes that were contacted for the read request and most importantly only repairs data when it is read.

Anti-entropy protocols are background mechanisms that run on every node and are responsible for keeping data synchronized between replica nodes, ensuring data convergence. Most data stores use Merkle Trees for this task, a topic already discussed previously.

MULTI-VALUE MEMORY In classic (single-value) memory, a read returns a single value. For shared-memory concurrent programming in multi-cores, even when weak memory models are offered, programmers have a toolset of synchronization primitives that can be used to, e.g., prevent data-races or enforce mutual-exclusion.

In a distributed setting aiming for high-availability, namely a distributed key-value store, such strong synchronization primitives are missing. Having the classic single-value memory API will result in lost updates, when several processes concurrently perform a read-compute-write update to the same location. This happens in most systems that offer causal consistency, such as COPS [36] and Eiger [37].

Dynamo [15] was seminal in exposing a multi-value memory API, in which a read returns the set of values concurrently written. This allow writing applications in which such concurrency is detected, and either some ad-hoc reconciliation is performed upon a read, or general CRDTs [54] with well defined merge operations are used. Dynamo, however, does not provide causal consistency, making life difficult for programmers.

PARTITIONING Consistent hashing [27] is one of the most popular algorithms employed in distributed data stores to partition keys across several nodes, since it randomly spreads the load across the nodes and it minimizes data migration when we change the cluster's size.

Riak for example, uses consistent hashing to assign keys to *virtual nodes (vnodes)*, which in turn are mapped to physical nodes. One physi-

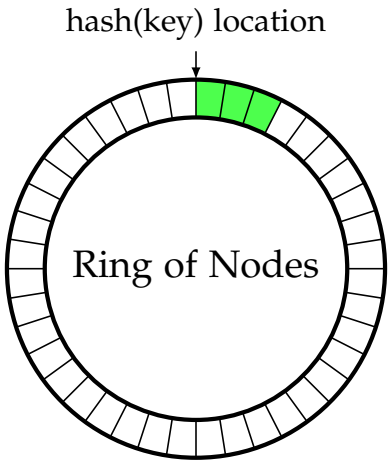


Figure 3: Logical Ring for Consistent Hashing with Replication Factor of 3.

cal node can have several vnodes. When a key is hashed, it gives a value that maps to a circular ring of vnodes. That key is replicated by the vnode responsible for that value in the hash, plus the next *N-1* vnodes in the ring, where N is the replication factor of that key.

2.4.2 *Causally Consistency Data Stores*

Causal Consistency is the strongest model that still enables high-availability, which makes it one of the most useful consistency models that are part of the broader "weakly-consistent" category. We will briefly describe some of the most well known causally consistent systems in the literature.

2.4.2.1 *COPS*

COPS and its variant COPS-GT [36] are both causally consistency geo-replicated data stores. The latter also provides read transactions to the client. Each data-center (DC) is a linearizable system with a complete copy of the database and causal consistent is only provided between data-centers. Each data version has a Lamport clock that is used to track explicit causal dependencies. However, it does not support concurrent updates without losing data, since it uses a physical timestamp to choose the *latest* version.

Data replication to other DCs blocks until all dependencies are met at the destiny location. This way the data between DCs is always causally consistent, but performance suffers. Inside a single DC, it avoids blocking to wait for dependencies, because a client session is only valid in one DC at a time, and since each DC is a linearizable system, local dependencies are always met.

2.4.2.2 *Eiger*

Eiger [37] is the successor of COPS that introduces a new column-family data model (instead of key-value from COPS), write transactions and instead of using data versions as dependencies, it uses operations as dependencies. It still suffers from the same problems as COPS, but has better performance due coalesce the dependency of multiple data items into one operations dependency, when transactions are used. Write transactions are achieved with a modified two-phase commit protocol that functions differently depending if it is running on the DC that accepted the operation initially or in a remote DC.

2.4.2.3 *Orbe*

Orbe [17] offers causal-consistency by using data versions as dependencies like COPS, but instead of Lamport clocks it uses version vectors that are encoded in a sparse matrix. It has a extension to the protocol that allows for read transactions that added physical timestamps to the dependency matrix. It does not support partial replication or concurrent writes with a multi-value API. Client sessions are also bound to the DC they are running. It also suffers from the same problem as COPS and Eiger regarding data replication, since it needs to apply updates in causal order, thus potentially blocking replication until dependencies arrive.

2.4.2.4 *GentleRain*

GentleRain [18] is the successor to the Orbe system by the same authors. The key difference is the usage of physical timestamps to track dependencies and enforce causal consistency. Each node stores a set

of objects that have a timestamp associated with the current timestamp. Then, each node keeps track of the *local stable timestamp* (LST), which means that objects with timestamps older than LST have been fully replicated. There is also a *global stable timestamp* (GST) for the entire system, computed by exchanging LSTs using a deterministic tree overlay (using node IDs total order) from leafs nodes to the root and back downs with the final GST. This information is used to garbage collected older objects kept around for transactions and is used to know when objects stored in remote DCs are safe to be read.

A key requirement for this system design is that physical must be monotonically increasing. Also, clock skew between nodes in the system must be kept to a minimum, since GentleRain relies on the GST value to garbage collect metadata a decide when certain objects are safe to be visible. They could have used Lamport clocks instead of timestamps, but logical clocks increment independently (usually increment by one for every write) which makes it difficult synchronize different logical clocks to compute the GST. Thus, using Lamport clocks would solve the clock skew issue, but would either introduce coordination, or the difference between the GST and all LST's would be greater, leading to objects taking longer to be visible to clients and metadata would also take longer to remove.

2.4.2.5 *SwiftCloud*

SwiftCloud [62] provides causal consistency with a focus on edge computing, where clients have a local process *scouts* with caching that act as a middle-man between the client and the server. These scouts can execute client operations without necessarily contacting the server. This is because objects are CRDTs which makes them mergeable and convergent, and therefore they are guaranteed to not fail (i.e. conflict) at the server, regardless of other clients operations. This also means that non-deterministic operations are not supported.

DCs exchange data using causal broadcast, which ensures that each DC applies new writes in their causal order, maintaining causal consistency. Since each DC is causally consistent, scouts can transparently

change to other DCs while maintaining causal consistency, because they cache the client operation results and they only make objects in other DC's visible to the client if they are K-replicated, where K can be configured by the system administrator.

2.4.2.6 *Bolt-On*

Bolt-on [8] implements a middleware between clients and a eventually consistent data store. The middleware enforces causal consistency for different clients, caching results of calls to the underlying data store. The interesting aspect of Bolt-on is that is can be implemented on top of existing eventually consistent data stores.

However, the implementation uses extensive metadata to provide the causal consistency semantics at the middleware layer. To mitigate this, they focus on the fact that applications can distinguish from explicit causality (only related operations at the application level should be written in the same causal session) and potential causality (every operation from a client is causally related), being the latter much smaller and producing less metadata.

2.4.2.7 *Lazy Replication*

Lazy Replication [31] is a technique developed by Ladin et al. to provide causal consistency in a fully replicated data store. This means that the entire state must fit in the smallest machine in the cluster. The system offers linearizable operations, and also has support for client-side replicas, which makes the garbage collection of causal metadata more difficult, since they are required to safely prune metadata but they can disconnect without warning. The garbage collection also relies on loosely-synchronized physical clocks. Clients can define themselves the dependencies of an operation using specific application logic. Each write operation is tagged with a Lamport clock and nodes keep track of the current local state using a version vector. The primary node responsible for each object must serialize a log to exchange data between nodes.

2.4.2.8 *PRACTI*

The PRACTI [9] system provides causal consistency with support partial replication, i.e., does do not have to store the entire state. However, one drawback is that any dependencies of an object must also reside on the same node, thus related data cannot be partitioned and the process partitioning data in some cases cannot be automatic (e.g. if data relationships are not known in advance).

Nodes can exchange information not in causal order, but each node serializes the exchange information for the objects that it is primary node in a log to exchange with other replicas. Concurrent writes are also not supported, relying in physical timestamps to choose the latest object. Another scalability problem with with PRACTI is that nodes maintain VVs with client IDs, which can grow unbounded.

2.5 DISCUSSION

Our focus on this thesis is in highly available distributed data stores, categorized as "AP" systems in the CAP terminology. Many data stores in this space fall in two subcategories: *eventually consistent* data stores, or *causally consistent* data stores. They share most of their architecture, but differing on the guarantees provided by their client API. In either case, the data store may or may not support the Dynamo-style multi-value API, regardless of the exact data model used by the data store (key-values, column-families, documents, etc.).

We will use a bottom-up approach on this thesis, to tackle the different challenges present in these systems. We will tackle first the correct causally tracking on multi-value eventually consistent data stores. Then we will research ways of improving the scalability of these systems, namely the space-complexity of logical clocks to track causality and the efficiency of anti-entropy protocols for systems that track causality.

Finally, we want to improve on those results and implement them on cross-key causally consistent data stores. However, most of the state-of-the-art data stores do not support the multi-value API necessary to

support write concurrency, which will also be a main part of our research.

3

DOTTED VERSION VECTORS

We present a new and simple causality tracking solution, Dotted Version Vectors (briefly introduced in [49]), that overcomes the limitations of traditional version vectors, allowing both scalable (using server ids) and fully accurate (representing same server concurrent writes) causality tracking. It achieves this by explicitly separating a new write event identifier from its causal past, which has the additional benefit of allowing causality checks between two clocks in constant time (instead of linear with the size of version vectors).

Besides fully describing Dotted Version Vectors (`DVV`), here we make two novel contributions. First, we propose a new container (DVV Sets or `DVVS`) that efficiently compacts a set of concurrent `DVV`'s in a single data structure, improving on two `DVV` limitations: (1) `DVVS` have a single common representation of causality, for all concurrent values, not linear in the number of values; (2) comparing and synchronizing two replica servers for each single key is linear with the number of concurrent values, instead of quadratic.

The other contribution is a general framework that clearly defines a set of functions that logical clocks need to implement to correctly track causality in eventually consistent systems. We implement both `DVV` and `DVVS` using this framework.

3.1 SYSTEM MODEL AND DATA STORE API

We consider a standard Dynamo-like key-value store interface that exposes two operations: `get`(key) and `put`$(key, value, context)$. `get` re-

turns a pair $(value(s), context)$, i.e., a value or set of causally concurrent values, and an opaque `context` that encodes the causal knowledge in the value(s). `put` submits a single value that supersedes all values associated to the supplied `context`. This `context` is either empty if we are writing a new value, or some opaque data structure returned to the client by a previous `get`, if we are updating a value. This `context` encodes causal information, and its use in the API serves to generate a *happens-before* [53] relation between a `get` and a subsequent `put`.

We assume a distributed system where nodes communicate by asynchronous message passing, with no shared memory. The system is composed by possibly many (e.g., thousands) clients which make concurrent `get` and `put` requests to server nodes (in the order of, e.g., hundreds). Each key is replicated in a typically small subset of the server nodes (e.g., three nodes), which we call the replica nodes for that key. These different orders of magnitude of clients, servers and replicas play an important role in the design of a scalable causality tracking mechanism.

We assume: no global distributed coordination mechanism, only that nodes can perform internal concurrency control to obtain atomic blocks; no sessions or any form of client-server affinity, so clients are free to read from a replica server node and then write to a different one; no byzantine failures; server nodes have stable storage; nodes can fail without warning and later recover with their last state in the stable storage.

As in this chapter we do not aim to track causality between different keys, in the remainder of the chapter we will focus on operations over a single key, which we leave implicit; namely, all data structures in servers that we will describe are per key. Techniques as in [41] can be applied when considering groups of keys and can introduce additional savings; this we leave for subsequent chapters.

3.2 CURRENT APPROACHES

To simplify comparisons between different mechanisms, we will introduce a simple execution example between clients Mary and Peter, and a

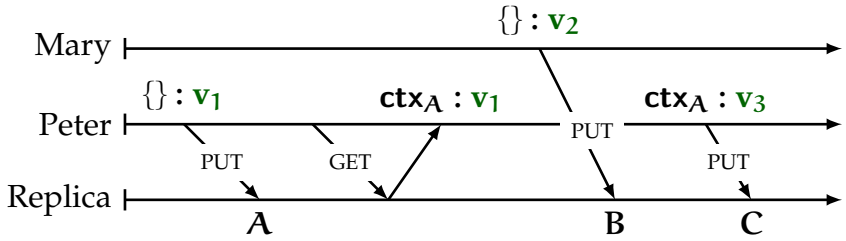


Figure 4: Example execution for one key: Peter writes a new value v_1 (**A**), then reads from Replica (**ctx$_A$**). Next, Mary writes a new value v_2 (**B**) and finally Peter updates v_1 with v_3 (**C**).

	LWW	CH	VV$_{client}$	VV$_{server}$	DVV	DVVS
A	17h00 : v_1	$\{r_1\} : v_1$	$\{(p,1)\} : v_1$	$\{(r,1)\} : \{v_1\}$	$((r,1),\{}) : v_1$	$\{(r,1,[v_1])\}$
ctx$_A$	{}	$\{r_1\}$	$\{(p,1)\}$	$\{(r,1)\}$	$\{(r,1)\}$	$\{(r,1)\}$
B	17h03 : v_2	$\{r_1\} : v_1$ $\{r_2\} : v_2$	$\{(p,1)\} : v_1$ $\{(m,1)\} : v_2$	$\{(r,2)\} : \{v_1,v_2\}$	$((r,1),\{}) : v_1$ $((r,2),\{}) : v_2$	$\{(r,2,[v_2,v_1])\}$
C	17h07 : v_3	$\{r_2\} : v_2$ $\{r_1,r_3\} : v_3$	$\{(m,1)\} : v_2$ $\{(p,2)\} : v_3$	$\{(r,3)\} : \{v_1,v_2,v_3\}$	$((r,2),\{}) : v_2$ $((r,3),\{(r,1)\}) : v_3$	$\{(r,3,[v_3,v_2])\}$

Table 1: The table shows the replica (r) state after write from Peter (p) and Mary (m), and the context returned by Peter's read. We use the $metadata : value(s)$ notation, except for DVVS which has its own internal structure.

single replica node. In this example, presented in Figure 4, Peter starts by writing a new object version v_1, with an empty context, which results in some server state A. He then reads server state A, returning current version v_1 and context ctx_A. Meanwhile, Mary writes a new version v_2, with an empty context, resulting in some server state B. Since Mary wrote v_2 without reading state A, state B should contain both v_1 and v_2 as concurrent versions, if causality is tracked. Finally, Peter updates version v_1 with v_3, using the previous context ctx_A, resulting in some state C. If causal relations are correctly represented, in state C we should only have v_2 and v_3, since v_1 was superseded by v_3 and v_2 is concurrent with v_3. We now discuss how different causality tracking approaches address this example, which are summarized in Table 1.

3.2.1 *Last Writer Wins*

In systems that enforce a Last Writer Wins (LWW) policy, such as Cassandra, concurrent updates are not represented in the stored state and only the last update prevails. Under LWW, our example would result in the loss of $\mathbf{v}_2$. Although some specific application semantics are compatible with a LWW policy, this simplistic approach is not adequate for many other application semantics. In general, a correct tracking of concurrent updates is essential to allow all updates to be considered for conflict resolution.

3.2.2 *Causal Histories*

Causal Histories (CH) [53] are simply described by sets of unique write identifiers. These identifiers can be generated with a unique identifier and a monotonic counter. In our example, we used server identifiers r, but client identifiers could be used as well. The crucial point is that identifiers have to be globally unique to correctly represent causality. Let id_n be the notation for the n^{th} event of the entity represented by *id*. The partial order of causality can be precisely tracked by comparing these sets under set inclusion. Two CH are concurrent if neither includes the other: $A \parallel B$ iff $A \not\subseteq B$ and $B \not\subseteq A$. CH correctly track causality relations, as can be seen in our example, but have a major drawback: they grow linearly with the number of writes.

3.2.3 *Version Vectors*

Version Vectors (VV) are an efficient representation of CH, provided that the CH has no gaps in each *id*'s event sequence. A VV is a mapping from identifiers to counters, and can be written as a set of pairs $(id, counter)$; each pair represents a set of CH events for that *id*: $\{id_n \mid 0 < n \leqslant counter\}$. In terms of partial order, $A \leqslant B$ iff $\forall (i, c_a) \in A \cdot \exists (i, c_b) \in B \cdot c_a \leqslant c_b$. Again, $A \parallel B$ iff $A \not\leqslant B$ and $B \not\leqslant A$. Whether client or server identifiers are used in VV has major consequences, as we'll see next.

3.2.4 *Version Vectors with Id-per-Client*

Version Vectors with Id-per-Client (VV_{client}) uses `VV` with clients as unique identifiers. An update is registered in a server by using the client identification issued in a `put`. This provides enough information to accurately encode the concurrency and causality in the system, since concurrent client writes are represented in the VV_{client} with different ids. However, it sacrifices scalability, since VV_{client} will end up storing the ids of all the clients that ever issued writes to that key. Systems like Dynamo try to compensate this by *pruning* entries in VV_{client} at a specific threshold, but it typically leads to false concurrency and further need for reconciliation. The higher the degree of pruning, the higher is the degree of false concurrency in the system.

3.2.5 *Version Vectors with Id-per-Server*

If causality is tracked with Version Vectors with Id-per-Server (VV_{server}), i.e., using `VV` with server identifiers, it is possible to correctly detect concurrent updates that are handled by different server nodes. However, if concurrent updates are handled by the same server, there is no way to *express* the concurrent values — *siblings* — separately. To avoid overwriting siblings and losing information (as in `LWW`), a popular solution to this is to group all siblings under the same VV_{server}, losing individual causality information. This can easily lead to false concurrency: either a write `context` causally dominates the server VV_{server}, in which case all siblings are deemed obsolete and replaced by the new value; or this new value must be added to the current siblings, even if some of them were in its causal past.

Using our example, we finish the execution with all three values $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$, when in fact $\mathbf{v}_3$ should have obsoleted $\mathbf{v}_1$, like the other causally correct mechanisms in Table 2 (expect for `LWW`).

With VV_{server}, false concurrency can arise whenever a client *read-write cycle* is interleaved with another concurrent write on the same server. This can become especially problematic under heavy load with many

clients concurrently writing: under high latency, if a read-write cycle cannot be completed without interleaving with another concurrent write, the set of siblings will keep on growing. This will make messages grow larger, the server load heavier, resulting in a positive feedback loop, in what can be called a *sibling explosion*.

3.3 DOTTED VERSION VECTORS

We now present an accurate mechanism that can be used as a substitute for classic version vectors (VV) in eventually consistent stores, while still using only one *Id* per replica node. The basic idea of *Dotted Version Vectors* (DVV) is to take a VV and add the possibility of representing an individual causal event — *dot*[1] — separate from the rest of the contiguous events. The dot is kept separate from the causal past and it globally and uniquely identifies a write. This allows representing concurrent writes, on the same server, by having different dots.

In our example from Figure 4, we can see that state B is represented with a unique dot for both $\mathbf{v}_1$ and $\mathbf{v}_2$, even-though they both were written with an equally empty context. This distinction in their dots is what enables the final write by Peter to correctly overwrite $\mathbf{v}_1$, since the context supersedes its dot (and DVV), while maintaining $\mathbf{v}_2$ which has a newer dot than the context. In contrast, VV_{server} loses this distinction gained by separating dots by grouping every sibling in one VV and thus cannot know that $\mathbf{v}_1$ is outdated by $\mathbf{v}_3$.

3.3.1 *Definition*

A DVV consists in a pair (d, v), where v is a traditional VV and the dot d is a pair (i, n), with i as a node identifier and n as an integer. The dot uniquely represents a write and its associated version, while the VV represents the causal past (i.e. its context). The causal events (or dots)

1 The term dot denotes an isolated event "over" a version vector, as a diacritic dot is placed over an "ı" to form an "i".

represented by a DVV can be generated by a function CH that translates logical clocks to causal histories (CH can be viewed as sets of dots):

$$\begin{aligned} \mathsf{CH}(((i,n),v)) &= \{i_n\} \cup \mathsf{CH}(v), \\ \mathsf{CH}(v) &= \bigcup_{(i,n)\in v} \{i_m \mid 1 \leqslant m \leqslant n\}, \end{aligned}$$

where i_n denotes the n^{th} dot generated by node i, and $\mathsf{CH}(v)$ is the same function but for traditional VV. With this definition, the CH $\{a_1, b_1, b_2, c_1, c_2, c_4\}$ that cannot be represented by VV, can now be represented by the DVV $((c,4),\{(a,1),(b,2),(c,2)\})$.

3.3.2 *Partial Order*

The partial order on DVV can be defined in terms of inclusion of CH; i.e.:

$$X \leqslant Y \iff \mathsf{CH}(X) \subseteq \mathsf{CH}(Y),$$

Given that each dot is generated as a globally unique event, the partial order on possible DVV values becomes:

$$((i,n),u) < ((j,m),v) \iff n \leqslant v[i] \wedge u \leqslant v,$$

where the traditional point-wise comparison of VV is used: $u \leqslant v \iff \forall_{(i,n)\in u}.\, n \leqslant v[i]$.

An important consequence of keeping the dot separate from the causal past is that, if the dot in X is contained in the causal past of Y, it means that Y was generated causally after X, thus Y also contains the causal past of X. This means that there is no need for the comparison of the VV component and the order can be computed as an $O(1)$ operation (assuming access to a map data structure in effectively constant time), simply as:

$$((i, n), u) < ((j, m), v) \iff n \leqslant v[i].$$

3.4 DOTTED VERSION VECTOR SETS

Dotted Version Vectors (DVV), as presented in the previous section, allow an accurate representation of causality using server-based ids. Still, a DVV is kept for each concurrent version: $\{(dvv_1, v_1), (dvv_2, v_2), \ldots\}$. We can go further in exploring the fact that operations will mostly handle sets of DVV, and not single instances.

We propose now that the set of $(dvv, version)$ for a given key in a replica node is represented by a single instance of a container data type, a *Dotted Version Vector Set* (DVVS), which describes causality for the whole set. DVVS factorizes out common knowledge for the set of DVV described, and keeps only the strictly relevant information in a single data structure. This results in not only a very succinct representation, but also in reduced time complexity of operations: the concurrent values will be indexed and ordered in the data structure, and traversal will be efficient.

3.4.1 *From a Set of Clocks to a Clock for Sets*

To obtain a logical clock for a set of versions, we will explore the fact that at each node, the set of DVV as a whole can be represented with a compact VV. Formally this invariant means that, for any set of DVV S, for each node id i, *all* dots for i in S form a contiguous range up to some dot. Note that we can only assume to have this invariant, if we follow some protocol rules enforced by our framework, described in detail in section 3.5.3.

Assuming this invariant, we obtain a logical clock for a set of (dvv, $version$) by performing a two-step transformation of the sets of versions. In the first step, we compute a single VV for the whole set — the *top vector* — by the pointwise maximum of the dots and VV in the DVV's;

additionally, for each DVV in the set, we discard the VV component. As an example, the following set:

$$\{(((r,4),\{(r,3),(s,5)\}),v_1),$$
$$(((r,5),\{(r,2),(s,3)\}),v_2),$$
$$(((s,7),\{(r,2),(s,6)\}),v_3)\},$$

generates the top vector $\{(r,5),(s,7)\}$ and is transformed to a set of *(dot, version)*:

$$\{((r,4),v_1),((r,5),v_2),((s,7),v_3)\}.$$

This first transformation has incurred in a loss of knowledge: the specific causal past of each version. This knowledge is not, however, needed for our purposes. The insight is that, to know whether to discard or not a pair (dot,version) (d,v) from some set when comparing with another set of versions S, we do not need to know exactly *which* version in S dominates d, but only that *some* version does; if version v is not present in S, but its dot d is included in the causal information of the whole S (which is now represented by the top vector), then we know that v was obsolete and can be removed.

In the *second step*, we use the knowledge that all dots for each server id, form a contiguous sequence up to the corresponding top vector entry. Therefore, we can associate a list of versions (siblings) to each entry in the top vector, where each dot is implicitly derived by the corresponding version position in the list. In our example, the whole set is then simply described as:

$$\{(r,5,[v_2,v_1]),(s,7,[v_3])\},$$

where the head of each list corresponds to the more recently generated version at the corresponding node. The first version has the dot

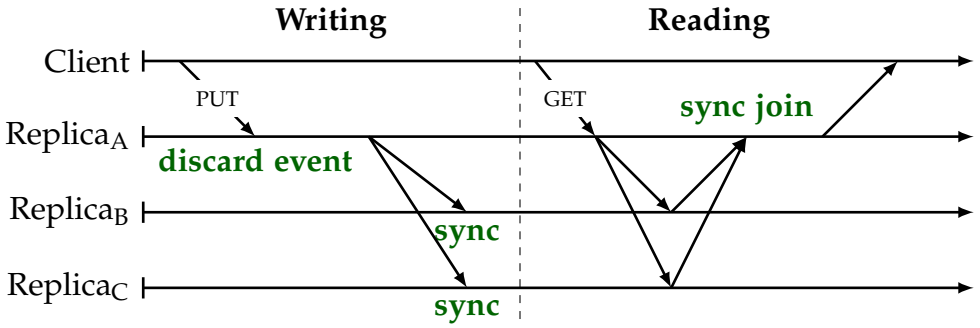


Figure 5: Generic execution paths for operations `get` and `put`.

corresponding to the maximum of the top vector for that entry, the second version has the maximum minus one, and so on.

3.4.2 *Definition*

A `DVVS` is a set of triples (i, n, l), each containing a server id, an integer, and a list of concurrent versions. It describes a set of versions and their dots, implicitly given by the position in the list. It also describes only the knowledge about the collective causal history, as given by the `VV` derived from the pairs (i, n).

3.5 USING DVV AND DVVS IN DISTRIBUTED KEY-VALUE STORES

In this section we show how to use logical clocks — in particular `DVV` and `DVVS`— in modern distributed key-value stores, to accurate and efficiently track causality among writes in each key. Our solution consists in a general workflow that a database must use to serve `get` and `put` requests. Towards this, we define a kernel of operations over logical clocks, on top of which the workflow is defined. We then instantiate these operations over the logical clocks that we propose, first `DVV` and then `DVVS`.

We support both `get` and `put` operations, performing possibly several steps, as sketched in Figure 5. Lets first define our kernel operations.

FUNCTION sync The function sync takes two sets of clocks, each describing a set of siblings, and returns the set of clocks for the siblings that remain after removing obsolete ones. It can have a general definition only in terms of the partial order on clocks, regardless of their actual representation: Equation 1.

FUNCTION join The join function takes a set of clocks and returns a single clock that describes the collective causal past of all siblings in the set received. An actual implementation of join is any function that corresponds to performing the union of all the events (dots) in the CH corresponding to the set, i.e., that satisfies Equation 2.

FUNCTION discard The discard function takes a set of clocks S (representing siblings) and a clock C (representing the context), and discards from S all siblings that are obsolete because they are included in the context C. Similar to sync, discard has a simple general definition only in terms of the partial order on clocks: Equation 3.

FUNCTION event The event function takes a set of clocks S (representing siblings), a clock C (representing the context) and a replica node identifier r; it returns a new clock to represent a new version, given by a new unique event (dot) generated at r, and having C in the causal past. An implementation must respect Equation 4.

$$\begin{aligned}
\mathsf{sync}(S_1, S_2) &= \{x \in S_1 \mid \nexists y \in S_2 . x < y\} \cup \\
&\quad \{x \in S_2 \mid \nexists y \in S_1 . x < y\} && (1)\\
\mathsf{CH}(\mathsf{join}(S)) &= \bigcup \{\mathsf{CH}(x) \mid x \in S\} && (2)\\
\mathsf{discard}(S, C) &= \{x \in S \mid x \not\leqslant C\} && (3)\\
\mathsf{CH}(\mathsf{event}(C, S, r)) &= \mathsf{CH}(C) \cup \{\mathsf{next}(C, S, r)\} && (4)
\end{aligned}$$

Function next denotes the next new unique event (dot) generated with r, which can be deterministically defined given C, S and r.

3.5.1 *Serving a Get*

Functions sync and join are used to define the `get` operation: when a server receives a `get` request, it may ask to a subset of replica nodes for their set of versions and clocks for that key, to be then "merged" by applying sync pairwise; however, the server can skip this phase if it deems it unnecessary for a successful response. Having the necessary information ready, it returns to the client both the values stripped from causality information and the context as a result of applying join to the clocks. sync can also be used at other times, such as anti-entropy synchronization between replica nodes.

3.5.2 *Serving a Put*

When a `put` request is received, the server forwards the request to a replica node for the given key, unless the server is itself a replica node. A non-replica node for the key being written could coordinate a `put` request if $\mathsf{VV}_{\mathsf{client}}$ were used, because it could use the *client Id* to update the clock and then propagate the result to the replica nodes. However, clocks using *server Ids* like $\mathsf{VV}_{\mathsf{server}}$, `DVV` and `DVVS` need the coordinating node to generate a unique event in the clock, using its own *Id*. Not forwarding the request to replica nodes, would mean that non-replica nodes Ids would be added to clocks, making them linear with the total number of servers (e.g. hundreds) instead of only the replica nodes (e.g. three).

When a replica node r, containing the set of clocks S_r for the given key, receives a `put` request, it starts by removing obsolete versions from S_r, using function discard, resulting in S'_r; it also generates a new clock u for the new version with event; finally, u is added to the set of non-obsolete versions S'_r, resulting in S''_r.

The server can then save S''_r locally, propagate it to other replica nodes and successfully inform the client. The order of these three steps depends on the system's durability and replication parameters. Each

replica node that receives S_r'', uses function sync to apply it against its own local versions.

For each key, the steps at the coordinator (discarding versions, generating a new one and adding it to the non-obsolete set of versions) must be performed atomically when serving a given `put`. This can be trivially obtained by local concurrency control, and does not prevent full concurrency between local operations on different keys. For operations over the same key, a replica can pipeline the steps of consecutive `put` for maximizing throughput (note that some steps already need to be serialized, such as writing versions to stable storage).

3.5.3 *Maintaining Local Conciseness*

As previously stated, both `DVV` and `DVVS` have a crucial invariant that servers must maintain, in order to preserve their correctness and conciseness:

Invariant 1. ***(Local Clock Conciseness)*** *Every key at any server has locally associated with it a set of version(s) and clock(s), that collectively can be logically represented by a contiguous set of causal events (e.g., represented as a VV).*

To enforce this invariant, we made two design choices: *(rule 1)* a server cannot respond to a `get` with a subset of the versions obtained locally and/or remotely, only the entire set should be sent; *(rule 2)* a coordinator cannot replicate the new version to remote nodes, without also sending all local concurrent versions (siblings).

Without the first rule, clients could update a key by reading and writing back a new value with a context containing arbitrary gaps in its causal history. Neither `DVV` nor `DVVS` would be expressive enough to support this, since `DVV` only supports one gap (between the contiguous past and the dot) and `DVVS` does not support any.

Without the second rule, `DVVS` would clearly not work, since writes can create siblings, which cannot be expressed separately with this clock. It could work with `DVV`, however it would eventually result in some server not having a local concise representation for a key (e.g., the

network lost a previous sibling), which in turn would make this server unable to respond to `get` without contacting other servers (see *rule 1*); it would degrade latency and in case of partitions, availability could also suffer.

3.5.4 *Dotted Version Vectors*

Functions sync and discard for DVV can be trivially implemented according to their general definitions, by using the partial order for DVV, already defined in Section 3.3.2.

We will make use of some two functions: function ids returns the set of identifiers of a pair from a VV, a DVV or a set of DVV; the maxdot function takes a DVV or set of DVV and a server id and returns the maximum sequence number of the events from that server:

$$\begin{aligned}
\mathsf{ids}((i,_)) &= \{i\},\\
\mathsf{ids}(((i,_),v)) &= \{i\} \cup \mathsf{ids}(v),\\
\mathsf{ids}(S) &= \bigcup_{s \in S} \mathsf{ids}(s).\\
\mathsf{maxdot}(r,((i,n),v)) &= \max(\{n \mid i = r\} \cup \{v[r]\}),\\
\mathsf{maxdot}(r,S) &= \max(\{0\} \cup \{\mathsf{maxdot}(r,s) \mid s \in S\}).
\end{aligned}$$

Function join returns a simple VV, which is enough to accurately express the causal information. Function event can be defined as simply generating a new dot and using the context C, which is already a VV, for the causal past.

$$\begin{aligned}
\mathsf{join}(S) &= \{(i,\mathsf{maxdot}(i,S)) \mid i \in \mathsf{ids}(S)\}.\\
\mathsf{event}(C,S,r) &= ((r,\mathsf{maxdot}(r,S)+1),C).
\end{aligned}$$

3.5.5 *Dotted Version Vector Sets*

With DVVS, we need to make slight interface changes: functions now receive a single DVVS, instead of a set of clocks; and event now inserts the newly generated version directly in the DVVS.

For clarity and conciseness, we will assume R to be the complete set of replica nodes ids, and any absent id i in a DVVS, is promoted implicitly to the element $(i, 0, [])$. We will make use of the functions: $\mathsf{first}(n, l)$, that returns the first n elements of list l (or the whole list if it has less than n elements, or an empty list for non-positive n); $|l|$ for the number of elements in l, $[x \mid l]$ to append x at the head of list l; and function merge:

$$\mathsf{merge}(n, l, n', l') = \begin{cases} \mathsf{first}(n - n' + |l'|, l), & \text{if } n \geqslant n', \\ \mathsf{first}(n' - n + |l|, l'), & \text{otherwise.} \end{cases}$$

Function discard takes a DVVS S and a VV C, and discards values in S obsoleted by C. Similarly, sync takes two DVVS and removes obsolete values. Function join simply returns the top vector, discarding the lists. Function event is now adapted to not only produce a new event, but also to insert the new value, explicitly passed as parameter, in the DVVS. It returns a new DVVS that contains the new value v, represented by a new event performed by r and, therefore, appended at the head of the list for r. The context is only used to propagate causal information to the top vector, as we no longer keep it per version.

$$\begin{aligned} \mathsf{sync}(S, S') &= \{(r, \max(n, n'), \mathsf{merge}(n, l, n', l')) \mid \\ &\quad r \in R, (r, n, l) \in S, (r, n', l') \in S'\}, \\ \mathsf{join}(S) &= \{(r, n) \mid (r, n, l) \in S\}, \\ \mathsf{discard}(S, C) &= \{(r, n, \mathsf{first}(n - C(r), l)) \mid (r, n, l) \in S\}, \\ \mathsf{event}(C, S, r, v) &= \{(i, n + 1, [v \mid l]) \mid (i, n, l) \in S \mid i = r\} \cup \\ &\quad \{(i, \max(n, C(i)), l) \mid (i, n, l) \in S \mid i \neq r\} \end{aligned}$$

		LWW	CH	VV_{client}	VV_{server}	DVV	DVVS
	Space	$\tilde{O}(1)$	$\tilde{O}(U)$	$\tilde{O}(C \times V)$	$\tilde{O}(R+V)$	$\tilde{O}(R \times V)$	$\tilde{O}(R+V)$
Time	**event**	–	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(V)$	$\tilde{O}(R)$
	join	–	$\tilde{O}(U \times V)$	$\tilde{O}(C \times V)$	$\tilde{O}(1)$	$\tilde{O}(R \times V)$	$\tilde{O}(R)$
	discard	–	$\tilde{O}(U \times V)$	$\tilde{O}(C \times V)$	$\tilde{O}(R)$	$\tilde{O}(V)$	$\tilde{O}(R+V)$
	sync	–	$\tilde{O}(U \times V^2)$	$\tilde{O}(C \times V^2)$	$\tilde{O}(R+V)$	$\tilde{O}(V^2)$	$\tilde{O}(R+V)$
	PUT	$\tilde{O}(1)$	$\tilde{O}(S_w \times U \times V^2)$	$\tilde{O}(S_w \times C \times V^2)$	$\tilde{O}(S_w \times (R+V))$	$\tilde{O}(S_w \times V^2)$	$\tilde{O}(S_w \times (R+V))$
	GET	$\tilde{O}(1)$	$\tilde{O}(S_r \times U \times V^2)$	$\tilde{O}(S_r \times C \times V^2)$	$\tilde{O}(S_r \times (R+V))$	$\tilde{O}(R \times V + S_r \times V^2)$	$\tilde{O}(S_r \times (R+V))$
	Causally Correct	✗	✓	✓	✗	✓	✓

Table 2: Space and time complexity, for different causality tracking mechanisms. U: updates; C: writing clients; R: replica servers; V: (concurrent) versions; S_r and S_w: number of servers involved in a GET and PUT, respectively.

3.6 COMPLEXITY AND EVALUATION

Table 2 shows space and time complexities of each causality tracking mechanism, for a single key. Lets consider U the number of updates (writes), C the number of writing clients, R the number of replica servers, V the number of concurrent versions (siblings) and S_w and S_r the number of replicas nodes involved in a `put` and `get`, respectively. Note that U and C are generally several orders of magnitude larger than R and V. The complexity measures presented assume effectively constant time in accessing or updating maps and sets. We also assume ordered maps/sets that allow a pairwise traversal linear on the number of entries.

`LWW` is constant both in time and space, since it does not track causality and ignores siblings. Space-wise, `CH` and VV_{client} do not scale well, because they grow linearly with writes and clients, respectively. `DVV` scales well given that typically there is little concurrency per key, but it still needs a `DVV` per sibling. From the considered clocks, `DVVS` and VV_{server} have the best space complexity, but the latter is not causally accurate.

Following our framework (Section 3.5), the time complexities are[2]:

2 For simplicity of notation, we use the big O variant: $\tilde{O}$, that ignores logarithmic factors in the size of integer counters and unique ids.

- `put` is $\tilde{O}(discard + event + S_w \times sync)$ and `get` is $\tilde{O}(join + S_r \times sync)$;
- *event* is effectively $\tilde{O}(1)$ for `CH`, VV_{client} and VV_{server}; is linear with V for `DVV`, because it has to check each value's clock; and is $\tilde{O}(R)$ for `DVVS` because it also merges the context to the local clock;
- *join* is constant for VV_{server}, since there is already only one clock; for `CH`, VV_{client} and `DVV` it amounts to merging all their clocks into one; for `DVVS`, *join* simply extracts the top vector from the clock;
- *discard* is only linear with V in `DVV`, because it can check the partial order of two clocks in constant time; as for `CH` and VV_{client}, they have to compare the context to every version's clock; VV_{server} and `DVVS` always compare the context to a single clock, and in addition, `DVVS` has to traverse lists of versions;
- *sync* resembles *discard*, but instead of comparing a set of versions to a single context, it compares two sets of versions. Thus, `CH`, VV_{client} and `DVV` complexities are similar to *discard*, but quadratic with V instead of linear. Since VV_{server} and `DVVS` have only one clock, the complexity of *sync* is linear on V.

3.6.1 *Evaluation*

We implemented both `DVV` and `DVVS` in Erlang [3], and integrated it with our fork of the NoSQL Riak datastore [4]. To evaluate the causality tracking accuracy of `DVVS`, and its ability to overcome the sibling explosion problem, we setup two equivalent 5 node Riak clusters, one using `DVVS` and the other VV_{server}.

We then ran a script[5] equivalent to the following: Peter (P) and Mary (M) write and read 50 times each to the same key, with read-write cycles interleaved (P writes then reads, next M writes then reads, in alternation). Figure 7 shows the growth in the number of siblings with every

3 https://github.com/ricardobcl/Dotted-Version-Vectors
4 https://github.com/ricardobcl/riak_kv/tree/dvvset
5 https://gist.github.com/ricardobcl/4992839

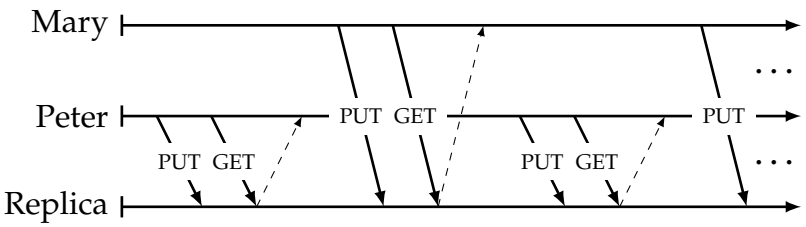


Figure 6: Peter and Mary interleave read-write cycles.

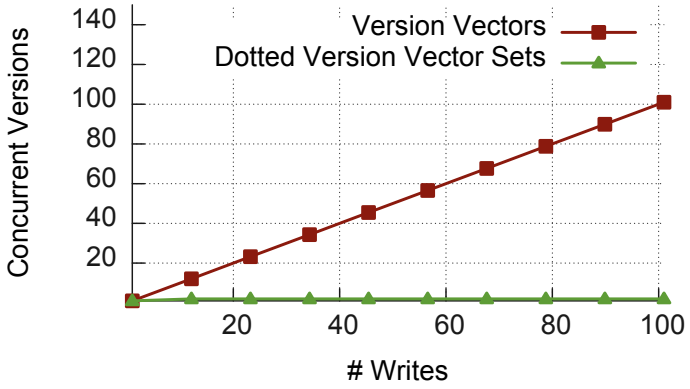


Figure 7: Results of running two interleaved clients with 50 writes each.

new write. The cluster with VV_{server} had an explosion of false concurrency: 100 concurrent versions after 100 writes. Every time a client wrote with the its latest context, the clock in the server was already modified, thus generating and adding a sibling. However, with `DVVS`, although each write still conflicted with the latest write from the other client, it detected and removed siblings that were causally older (all the siblings present at the last read by that client). Thus, the cluster with `DVVS` had only two siblings after the same 100 writes: the last write from each client.

Finally, `DVVS` has already seen early adoption in the industry, namely in Riak, where it is the default logical clock mechanism in the latest release. As expected, it overcame the sibling explosion problem that was affecting real world Riak deployments, when multiple clients wrote on the same key.

3.7 DISCUSSION

We have presented Dotted Version Vectors, a novel solution for tracking causality among update events. The base idea is to add an extra isolated event over a causal history. This is sufficiently expressive to capture all causality established among concurrent versions (siblings), while keeping its size linear with the number of replicas.

We then proposed a more compact representation — Dotted Version Vector Sets — which allows for a single data structure to accurately represent causal information for a set of siblings. Its space and time complexity is only linear with the number of replicas plus siblings, better than all current mechanisms that accurately track causality.

Finally, we introduced a general workflow for requests to distributed data stores. It abstracts and factors the essential operations that are necessary for causality tracking mechanisms. Using `DVV` with its system workflow, at most a single update event that is outside the VV is needed, and thus a single *dot* per version is enough. `DVVS` goes further, by condensing all causal information in a VV, while being able to keep multiple implicit *dots*. This ensures just enough expressiveness to allow any number of concurrent clients and still avoids the size complexity of encoding a generic non sequential CH. Additionally, by isolating the dot that identifies the version, causality can be checked in $O(1)$ time for `DVV` instead of $O(n)$ time.

One could be led to think that the conciseness obtained using server-based ids would contradict Charron-Bost minimality result [13], which states that vector clocks `VC` [19, 42] are the most concise characterization of causality among process events. Such is not the case because, not only the problems addressed by `VC` and `VV` are different [2], but essentially because the present scenario does not involve direct client-to-client interaction: all interactions are intermediated by servers.

4

NODE-WIDE INTRA-OBJECT CAUSALITY MANAGEMENT

In this chapter we introduce a novel node-wide logical clock framework, Node-wide Dot-based Clocks (NDC), overcoming three fundamental limitations of the state of the art: (1) minimize the metadata per key necessary to track causality, avoiding its growth even in the face of node churn; (2) correctly and durably delete keys, with no need for tombstones; (3) offer a lightweight anti-entropy mechanism to converge replicated data, avoiding the need for Merkle Trees.

We evaluate DottedDB, a Dynamo-like key-value store which we implemented, based on the NDC framework, against MerkleDB, an otherwise identical database, except that it uses standard per-key logical clocks and Merkle Trees for anti-entropy, to ensure a fair comparison with the state-of-the-art and precisely measure the impact of the novel approach. Results show that: causality metadata per object always converges rapidly to only one id-counter pair; distributed deletes are correctly achieved without global coordination and with constant metadata; divergent nodes are synchronized faster, with less memory-footprint and with less communication overhead than using Merkle Trees.

4.1 SYSTEM OVERVIEW

4.1.1 *System Model*

The database is composed by a set of nodes, each with its own storage. Each node has a unique id and can only communicate with other nodes via asynchronous message passing. Messages can be lost and reordered. Nodes can crash and restart with stable storage, or can fail entirely and be replaced by a new node with a new id and an empty storage.

4.1.2 *Partial Replication*

Objects are replicated across a set of nodes. The number of replicas can be customized across the entire server or on a per-object basis. It is typically much smaller (e.g., 3) than the number of nodes (e.g., 100). Nodes that share replicas of some object are called *peer nodes*. In general, the common set of objects replicated by two peer nodes is only a small subset of the objects stored at either one.

4.1.3 *Client API*

The database is a key-value store, where objects are accessed through their key. A client can issue requests to any node in the server. If the contacted node does not hold a replica of the requested key, it forwards the request to one of the replica nodes for that key. These operations are available:

(values, context) ← **g**et (key)
bool ← **p**ut (key, value, context)
bool ← **d**elete (key, context)

When the client fetches an object by key, a list of objects is returned, reflecting possible concurrent updates, together with an opaque causal context. This causal context plays a role in maintaining the causal history of individual objects by allowing the client to link a get to a sub-

sequent update operation, either put or delete, which takes the causal context as an extra parameter.

Typically, a client wanting to perform a *read-modify-update* operation will perform a get, modify the value(s) returned and issue a put, passing the new value together with the causal context from the get.

4.2 NODE-WIDE DOT-BASED CLOCKS FRAMEWORK

Node-wide Dot-based Clocks (NDC) is a general framework for distributed, partitioned and replicated databases. It assumes a master-less collection of nodes, any of which can handle client requests, without distributed locking or global coordination. It aims to: (a) reduce to a minimum the amount of causality metadata stored per object, even with node churn; (b) provide a *causally-safe* way to delete objects with no need for tombstones; (c) provide a lightweight anti-entropy protocol exploiting the logical clocks. To achieve this, it makes use of some key ideas:

EVERY UPDATE HAS A GLOBALLY UNIQUE IDENTIFIER (DOT) Every time a node coordinates a client request that updates a local object (including deletes), it generates a new globally unique identifier, by pairing the node id with a node-wide monotonically increasing counter; we call this pair a *dot*. Since every version of an object in the system has a unique dot, nodes can summarize their knowledge of updates in a single data structure: the *node clock*. It contains all dots generated locally or received from peer nodes via replication or anti-entropy.

OBJECT METADATA MIGRATES TO THE NODE CLOCK Since the node clock summarizes the local storage history, causality metadata for each object version (i.e., dots representing its causal past) can eventually be omitted when saving to storage or when sending to another node, using the *strip* operation. When an object is fetched from storage, its causal past can be recovered through the node clock, using the *fill* operation.

CHURN RATE DOES NOT AFFECT OBJECT METADATA SIZE This *fill-strip* mechanism allows metadata in objects to remain effectively constant in size, even when new nodes keep entering the system to replace old nodes over time. This is because the dots from retired nodes are eventually included in the node clock and therefore stripped from objects.

THE NODE CLOCK IS THE DELETE TOMBSTONE In distributed data stores without coordination, client deletes only remove the payload of an object, leaving the causal metadata as a tombstone. This is done to avoid anomalies, such as receiving an older version of a deleted object via replication or anti-entropy, which will make the object reappear or even supersede recent writes.

However, the node clock can act as the tombstone for all deleted objects, since it eventually summarizes all metadata. Thus, deleted objects can be safely removed from storage, since they will be restored (i.e. filled) to the corresponding tombstone when read.

NODES CAN SYNCHRONIZE BY COMPARING NODE CLOCKS The anti-entropy protocol responsible for detecting and repairing obsolete data, can now simply compare node clocks, to learn which dots from one node are missing from another node.

4.2.1 *The Node Clock*

The node clock represents which update events this node has *seen*, directly (coordinated by itself) or transitively (received from others). In abstract, it represents the set of dots corresponding to those updates.

Concretely, the node clock groups dots per peer node, factoring out the node id part from the dots. Also, each set of counters associated with a node id, can be greatly compacted by exploiting the fact that dots are generated with consecutive counters. For each node, the node clock represents the set of counters in two parts: a *base* counter representing

the contiguous sequence starting from 1 (as in Version Vectors), and a set of non-contiguous counters.

Since the latter typically represents a small range of dots (the gaps in non-contiguous sets are filled in anti-entropy runs), it can be efficiently encoded as a bitmap, as in our implementation in DottedDB.

4.2.2 *Per-Object Clock*

An object internally encodes a logical clock by tagging every value (there can be multiple concurrent values) with a dot and storing causality information about all current and past versions also as dots. We call the former *versions* and the latter *causal context*. Dots are removed (stripped) from the causal context if they are included in the node clock. The dot in a version is never removed, since it is used to test if another object obsoletes that version.

Because dot generation is per-node instead of per-key, it is unlikely that dots in the causal context are contiguous. To solve this issue, we will use the notion of an *extrinsic* set, first defined by the authors of [41]. The following definition improves and generalizes the original definition (note that an event can be seen as a write made to a particular key):

Definition 4.2.1 (Extrinsic)**.** A set of events E_1 is said to be *extrinsic* to another set of events E_2, if the subset of E_1 events involving keys that are also involved in events from E_2, is equal to E_2.

This definition means that all gaps can be filled with extra dots, producing the *extrinsic* set of the original. Those extra dots are from versions of other keys, since an object containing a version with a dot (n, c), must have seen all prior versions coordinated by node n with a dot smaller than (n, c). Thus, the context can be represented only by the largest dot per node id, like a Version Vector, without sacrificing correctness.

4.2.3 *Node State*

The NDC framework requires each node to maintain five data-structures:

1. Node Clock (NC): all dots from current and past versions *seen* by this node;
2. Dot-Key Map (DKM): maps dots of locally stored versions to keys. This is required by the anti-entropy protocol to know which key corresponds to a missing dot that needs to be sent to another node. Entries are removed when dots are known by every peer node;
3. Watermark (WM): a cache of node clocks from every peer (including itself). It is used to know when a dot is present in all peers, enabling the removal of that entry in the DKM. It is updated in every anti-entropy round, taking advantage of the node clock exchange. In practice, only the base counter of every node clock entry is saved, resulting in a more compact representation as a matrix, although slightly delaying the garbage collection of DKM;
4. Non-Stripped Keys (NSK): the keys of local objects with a non-empty causal context. When an object is saved to storage, it may have entries in the causal context that are not included in the node clock. To guarantee that every object is eventually stripped of its causal context, this list is periodically iterated to check if the causal context can be completely removed;
5. Storage (ST): maps keys to objects.

The definition of these data-structures is as follows:

$$
\begin{aligned}
\mathsf{NC} &: \mathbb{I} \hookrightarrow \mathcal{P}(\mathbb{N}) \\
\mathsf{DKM} &: (\mathbb{I} \times \mathbb{N}) \hookrightarrow \mathbb{K} \\
\mathsf{WM} &: \mathbb{I} \hookrightarrow \mathbb{I} \hookrightarrow \mathbb{N} \\
\mathsf{NSK} &: \mathcal{P}(\mathbb{K}) \\
\mathsf{ST} &: \mathbb{K} \hookrightarrow \mathsf{Object} \\
\mathsf{Object} &: ((\mathbb{I} \times \mathbb{N}) \hookrightarrow \mathbb{V}) \times (\mathbb{I} \hookrightarrow \mathbb{N})
\end{aligned}
$$

Algorithm 1: Client API at Node i.

```
1  procedure GET(k : 𝕂, quorum[= 1] : ℕ):
2      O := ∅
3      parallel for j ∈ replica_nodes(k) do
4          O := O ∪ rpc(j, fetch, ⟨k⟩)
5      await(size(O) ⩾ quorum)
6      (vers, cc) := (∅, ∅)
7      for o ∈ O do
8          (vers, cc) := merge((vers, cc), o)
9      return (ran(vers), cc)
10
11 procedure PUT(k : 𝕂, v : 𝕍, cc : 𝕀 ↪ ℕ):
12     c := max(NC_i[i]) + 1
13     NC_i[i] := NC_i[i] ∪ c
14     ver := ((i, c), v)
15     o := update(k, ({ver}, cc))
16     for j ∈ replica_nodes(k) do
17         async_rpc(j, update, ⟨k, o⟩)
18
19 procedure DELETE(k : 𝕂, cc : 𝕀 ↪ ℕ):
20     PUT(k, null, cc)
```

4.2.4 *Serving Client Requests*

Algorithm 1 describes the three operations are available to a client: *GET*, *PUT*, and *DELETE*.

GET To read an object, the client specifies the key and optionally the quorum size for the number of replicas to fetch. Any node can coordinate a read; it first requests replica nodes for that key; when it obtains a sufficient number of replicas, it merges them; the resulting causal context is returned, along with all concurrent values.

PUT The coordinator node generates a new dot that together with the new value, forms the new version of this object. That version, together with the client context (with the new dot) forms a temporary object that is used to update the local object, merging them. Finally, the object is sent to other nodes that replicate that key.

Algorithm 2: Strip and Fill Operations at Node i.

```
1  function strip((vers, cc) : Object, NC : 𝕀 ↪ 𝒫(ℕ)):
2      for j ∈ dom(cc) do
           // function base returns the greatest counter b in some collection C, where
              ∀i ∈ [1, b]. i ∈ C
3          if cc[j] ⩽ base(NC[j]) then
4              cc := {j} ⩤ cc
5      return (vers, cc)
6
7  function fill(k : 𝕂, (vers, cc) : Object, NC : 𝕀 ↪ 𝒫(ℕ)):
8      for j ∈ replica_nodes(k) do
9          cc[j] := max(cc[j], base(NC[j]))
10     return (vers, cc)
```

DELETE The DELETE API is exactly like a write, but without a new value. The new dot is associated with a null value and the PUT operation is called. It is important to note that if the client context does not include the dot of some locally stored version, such version will not be deleted. This is the desired behavior because such version is causally concurrent with the delete. This respects causality and avoids anomalies, like clients unknowingly deleting a concurrent update from another client, or a slowly propagated delete removing future object updates.

4.2.5 *Auxiliary Operations*

The already discussed fill and strip operations are defined in Algorithm 2. There are four other auxiliary operations defined in Algorithm 3: fetch, store, update, merge.

Reading an object with fetch fills the context with the current node clock base, restoring causality information. The restored context can be larger than the original one without affecting correctness, because all new causal information is either from older versions of this key or from dots of others keys.

The store operation first strips the object. Then, if the causal context is empty and there are only *null* values, the object is removed from storage; otherwise, the object is saved to storage. In addition, it: (a)

Algorithm 3: Auxiliary Operations at Node i.

```
1  function fetch(k : 𝕂):
2      return fill(k, ST_i[k], NC_i)
3
4  procedure store(k : 𝕂, o : Object):
5      (vers, cc) := strip(o, NC_i)
       // remove object if no values and cc is empty
6      if {val ∈ ran(vers) | val ≠ null} = ∅ ∧ cc = ∅ then
7          ST_i := {k} ⩤ ST_i
8      else ST_i[k] := (vers, cc)
9      for (j, c) ∈ dom(vers) do
10         NC_i[j] := NC_i[j] ∪ {c}
11         DKM_i[(j, c)] := k
12     if cc = ∅ then NSK_i := NSK_i \ {k}
13     else NSK_i := NSK_i ∪ {k}
14
15 function merge((v_1, cc_1) : Object, (v_2, cc_2) : Object):
16     v := v_1 ∩ v_2
17     v_1 := {(dot, val) ∈ v_1 | dot ∉ cc_2}
18     v_2 := {(dot, val) ∈ v_2 | dot ∉ cc_1}
19     return (v ∪ v_1 ∪ v_2, cc_1 ∪ cc_2)
20
21 function update(k : 𝕂, o : Object):
22     (vers, cc) := merge(o, fetch(k))
       // make sure that the context covers the current version dots
23     for (j, c) ∈ dom(vers) do
24         cc[j] := max(cc[j], c)
25     store(k, (vers, cc))
26     return (vers, cc)
```

adds all version dots to the node clock and to the dot-key map, (b) adds the key to the non-stripped key set if the causal context is not empty, or removes the key from the set otherwise.

The merge function takes two objects, and returns a new object with the causal contexts merged (taking the maximum counter for common node ids) and the versions of each object not obsoleted by the other. An object version (dot, val) is obsoleted by another object $(vers, cc)$, if $(dot, val) \notin vers$ and $dot \in cc$. Finally, the update operation merges the receiving object with the local object and then stores and returns the result.

Algorithm 4: Anti-Entropy Protocol at Node i.

```
1  process anti_entropy():
2      loop forever
3          j := random(peers(i) \ {i})
4          async_rpc(j, sync_clock, ⟨i, NC_i⟩)
5          sleep(sync_interval)
6
7  procedure sync_clock(p : 𝕀, NC : 𝕀 ↪ 𝒫(ℕ)):
8      K := { k | ((j, c), k) ∈ DKM_i ∧ c ∉ NC[j] ∧ p ∈ replica_nodes(k) }
9      O := { (k, ST_i[k]) | k ∈ K }
10     async_rpc(p, sync_repair, ⟨i, NC_i, O⟩)
11
12 procedure sync_repair(p : 𝕀, NC : 𝕀 ↪ 𝒫(ℕ), O : 𝕂 × Object):
13     for (k, o) ∈ O do
14         update(k, fill(k, o, NC))
       // merge p's node clock entry to close gaps
15     NC_i[p] := NC_i[p] ∪ NC[p]
16     update_watermark(p, NC)
17     gc_dkm()
18
19 procedure update_watermark(p : 𝕀, NC : 𝕀 ↪ 𝒫(ℕ)):
       // update the WM with new i and p clocks
20     for j ∈ dom(NC) ∩ peers(i) do
21         WM_i[p][j] := max(WM_i[p][j], base(NC[j]))
22     WM_i[i][p] := base(NC_i[p])
23
24 procedure gc_dkm():
25     for (j, c) ∈ dom(DKM_i) do
26         if min({ WM_i[p][j] | p ∈ peers(j) }) ⩾ c then
27             DKM_i := {(j, c)} ⩤ DKM_i
```

4.2.6 *Background Tasks*

There are two background processes running at every node: the *anti-entropy* and the *causality stripping*. Both assume the definition of the function peers(i) that returns the node IDs of all peers of node i, including i.

ANTI-ENTROPY Algorithm 4 describes the anti-entropy background process running in every node. Periodically, the anti-entropy process in a node i chooses a random peer j to sync with, and sends its node clock.

Algorithm 5: Causality Stripping at Node i.

```
1 process strip_causality():
2     loop forever
3         for k ∈ NSK_i do store(k, ST_i[k])
4         sleep(strip_interval)
```

Node j collects all keys whose dots in the dot-key map are not present in node i history, while ignoring keys that are not replicated by i. It then fetches all local objects corresponding to those keys and sends them to i, along with all dots in j's history that were generated locally. Every object is read directly from storage without being filled, to save bandwidth; they are later filled at node i.

Upon receiving the missing objects, i updates all local objects with the received information. Additionally, it: (a) merges j's node clock entry into i's node clock, closing any gap from dots of j not replicated by i; (b) updates the watermark with the base of i's and j's node clock (using the update_watermark procedure); (c) removes any entry in the dot-key map, if the dot is known by all peers (using the gc_dkm procedure)[1].

CAUSALITY STRIP Algorithm 5 defines the periodic process run at every node to strip causality from objects. Periodically, the node iterates the non-stripped keys, reading each one directly from storage and storing them back. The store operation already takes care of the stripping and updates the NSK accordingly.

4.3 FAULT TOLERANCE

NDC makes few system assumptions, in order to tolerate as many failures as possible. Temporary network partitions or message loss simply delay the rate of convergence to a consistent state, since anti-entropy eventually repairs replicated data. Node churn temporarily increases metadata in all structures, since both the old and new node entries must be maintained, until the old entries can be safely removed from

1 The last two functions could be separated in two different processes with their own interval.

every structure. The node clock maintains the entire node history, including for retired nodes, in a compact way (retired nodes eventually are represented as entries in a version vector).

4.3.1 *Transitive Anti-Entropy Repair*

Tracking every peer's state in the watermark and keeping dot-key pairs even for objects coordinated by other peers, enables nodes to transitively exchange and repair objects with each other. If communication between some peers A and B is down, or they are not online at the same time, they can still be kept synchronized if a third peer C can communicate with both.

4.3.2 *Node Failures*

All NDC data structures and objects are updated in memory and periodically saved to durable storage to maintain consistency. Objects are saved first and if no failures occurred, then the node clock and other auxiliary structures are saved *atomically* to disk. Whether a node restarts and only loses its in-memory state, or also loses its storage, it should always obtain a new node id before resuming activity. Node ids must enable peers to identify the new node as a replacement for the old one. This can be done by having node ids made up of two components: one which identifies the key space covered and another being a globally unique id, over a total order.

A node with a new node id can immediately serve new client requests, since new dots are guaranteed to be globally unique. In the background, the node performs normal anti-entropy with all peers to recover any missing data.

When a node detects a new node id being used, it will start tracking how many rounds of anti-entropy were done for each peer. After performing anti-entropy with all of them, there cannot be missing objects (and corresponding dots) from the old node. Therefore, any gaps in the local node clock from the old node entry, are not relevant anymore and

they can be closed as if an anti-entropy run with the old node had been performed. This leaves a single integer in the node clock entry, and it guarantees that objects can always be stripped (or deleted form storage), even under node churn.

If a retired node comes back online and is allowed to perform anti-entropy with its peers, this can possibly result in some objects from the retired node to be deleted. The reason is that some dots could have been manually inserted into node clocks from current peers (as explained above), without having the corresponding objects locally stored. Since missing dots that are not stored locally are considered deletes, an anti-entropy round would delete those object in the retired node. Therefore, if a new node is added to the cluster, it is safe to assume that any data that only the old node had is lost, even if it comes back online.

Upon detecting a new node id, the node must also add a new node id entry to every relevant node clock in the watermark, as soon as possible. This is because computing the minimum common knowledge depends on having information about every participant, in this case all current peers. The old peer entry can be removed from the watermark as soon as there are no more dots from that old node in the dot-key map, since that is the purpose of the watermark.

4.4 EXPERIMENTAL EVALUATION

To evaluate NDC against the state of the art, we implemented two distributed databases: DottedDB and MerkleDB. They share the exact same codebase, but diverge in the way they handle anti-entropy, object causality and distributed deletes.

4.4.1 *DottedDB*

We implemented a distributed database named DottedDB that uses the NDC framework both as its anti-entropy mechanism and to respect per-object causality.

DottedDB was implemented in Erlang, using the distributed systems framework Riak Core [29]. It runs on a cluster of physical machines, each having multiple instances of an abstraction called a virtual node (vnode). Each vnode is completely independent and isolated, with its own storage and memory, much like our definition of nodes in the system model. They only communicate via message passing and run client requests sequentially. Vnodes are totally ordered in a ring and assigned to physical machines sequentially.

Data is replicated with consistent hashing [27]: the hash of a key maps to a specific ring position, and then the key/object is replicated to the next n vnodes in the ring, where n is the desired replication factor.

4.4.2 *MerkleDB*

We also implemented MerkleDB, a clone of DottedDB in every way, except that it uses Merkle Trees for its anti-entropy mechanism, and Dotted Version Vectors as the logical clock to track each object causality. This mimics the current industry state of the art for distributed key-value stores that track causality, as in Riak 2.0 and later versions.

4.4.3 *Configuration*

We ran both DottedDB and MerkleDB in a cluster of 5 physical machines, divided in 64 independent vnodes. An additional machine was used to ran YCSB [14], a benchmark tool for NoSQL distributed databases. Every run is made after a loading phase of the entire key-set. All machines used for these experiments have an Intel i3 CPU at 3.1GHz, 8GB of main memory, a 7200 RPM SATA disk, and are interconnected by a switched Gigabit network.

4.4.4 *Object Logical Clock*

In both systems, we used a logical clock to track object causality. MerkleDB's clock is a Dotted Version Vector, while DottedDB uses NDC for this pur-

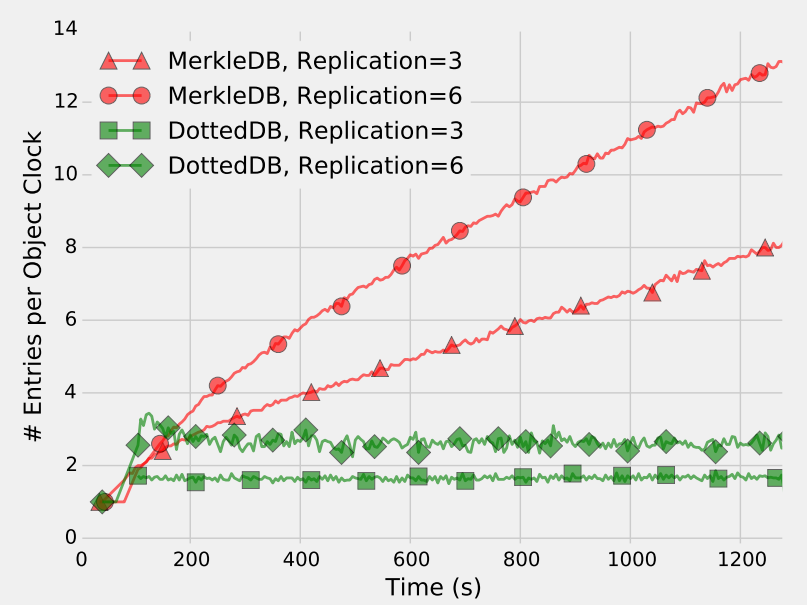


Figure 8: Average number of entries in object clocks written to storage, for two different replication factors, with node churn.

pose. We'll evaluate the scalability of the logical clock per object, how fast DottedDB can strip the clock and how that translates to actual distributed deletes.

4.4.4.1 *Object Clock Scalability*

Figure 8 shows the average number of entries in object clocks written to store over time. To simulate node churn, one node is replaced every 4 seconds. (This aggressive churn rate allows a short run to depict what would normally happen over a longer period.) Update requests are issued 150 times per second for 5000 keys, and we take a measure every time we write objects to storage.

MerkleDB never removes entries from the object clock; therefore, this number is proportional to the number of nodes that have ever been replica nodes for the key. Figure 8 clearly shows that with node churn, the average size of the object clock keeps growing, with new updates adding the new node ids to the clock.

In DottedDB, even under an aggressive churn rate and continuous updates over the key range, we can see that with a replication factor of 3 we have between 1 and 2 entries per clock written to storage on

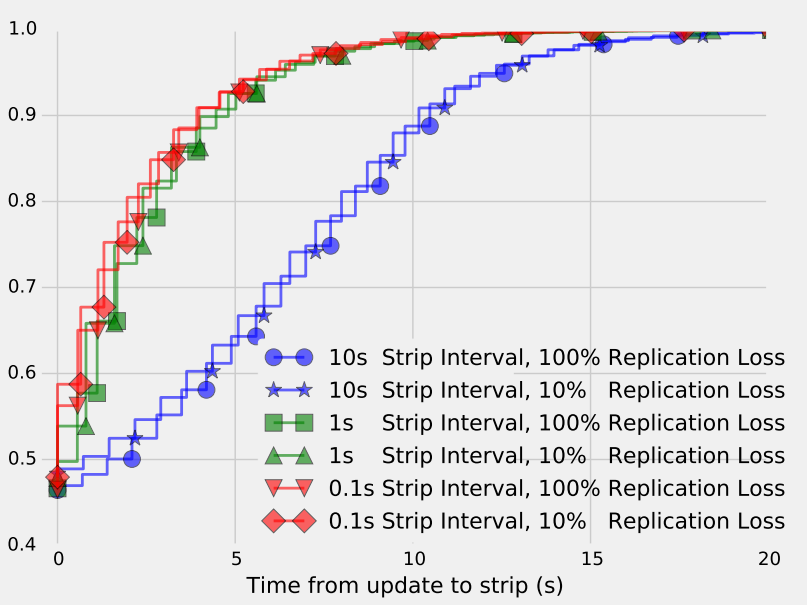


Figure 9: CDF (Cumulative Distribution Function) of time needed to strip the causal past in an object's clock, after the update at the coordinating node.

average. This figure grows to an average of 3 entries per object for a replication factor of 6. Importantly, this figures do not grow over time. New node ids from node churn do not pose a problem; entries are summarized by the node clock shorty after.

4.4.4.2 *Strip Latency*

Figure 9 shows the cumulative distribution of the time it takes for an object to have its entire context stripped in DottedDB, for a *sync interval* (time between anti-entropy runs) of 100 ms. We use different *strip intervals* (the time between each attempt to strip objects), either identical (0.1 sec) or slower (1 and 10 sec) than the sync interval; and two failure rates of replication messages when serving client requests (either 10%, or 100% – in which case replica propagation is only by anti-entropy). Even with a strip interval of 10 sec, after 20 sec almost all objects are stripped. Strip intervals of 100 ms or 1 sec give identical results: 90% of objects are stripped in less than 5 sec. Replication message loss when serving client requests did not have significant effect.

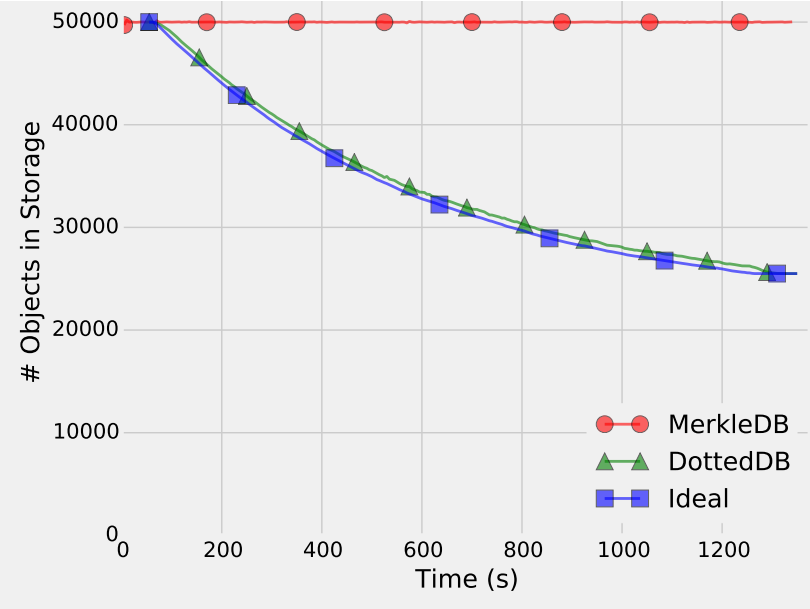


Figure 10: Number of objects in storage over time. Initially 50 000 objects, serving 100 ops/s, 50% writes and 50% deletes.

4.4.4.3 *Distributed Deletes*

MerkleDB does not remove the logical clock associated with an object that was deleted, keeping it stored as a *tombstone*. In DottedDB, as soon as we strip the object clock and no value remains, the entry can be safely removed from storage; i.e., remaining causally correct (due to the node clock) without the overhead of storing tombstones.

Figure 10 shows the total number of objects stored in a system pre-populated with 50 000 objects and serving 100 requests per second, 50% updates and 50% deletes. DottedDB correctly removes entries, without global coordination, in very little time. There is only a small delay between DottedDB and the *Ideal* scenario (immediate removal from storage). We have observed this delay to be proportional to the strip interval used, which in this run was 2.5 seconds.

Figure 11 show the cumulative distribution for the delay between a delete request and the actual removal from storage. Unsurprisingly, it is identical to Figure 9, since the pre-condition to remove an object is to strip its clock.

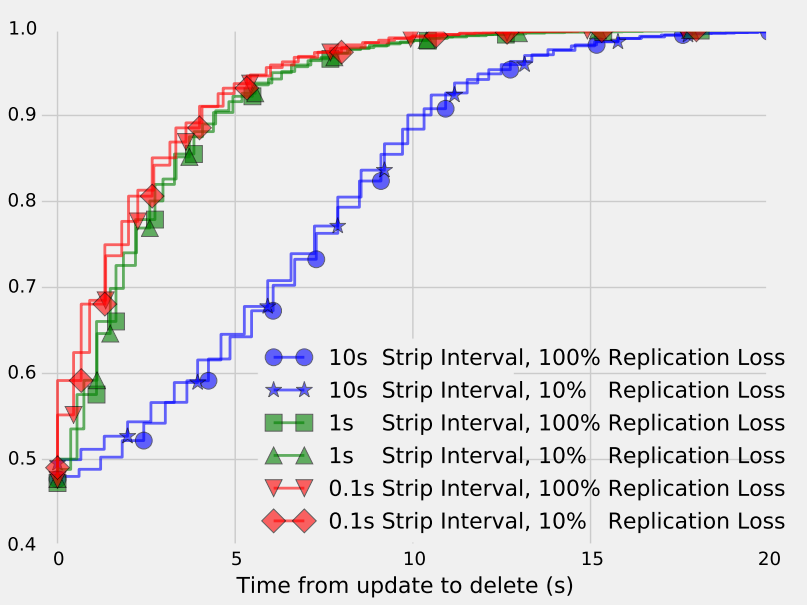


Figure 11: CDF of time it takes to remove an object from storage, since the delete was issued at the coordinating node.

4.4.5 *Anti-Entropy*

We compare the anti-entropy used by MerkleDB and DottedDB in four aspects: (1) *hit ratio*: percentage of the objects exchanged that were truly needed; (2) *node metadata* used by each mechanism; (3) *network usage* while performing anti-entropy; (4) *replication latency*: the delay between an object being written by the coordinator and the object being stored in all replicas.

We evaluated 20 minute runs in a database with 500 000 keys, doing 2500 updates/s (each client *update* includes reading an object before writing back an updated version), in different scenarios according to a combination of three parameters, each being either *H*igh or *L*ow, with the values in Table 3: objects per Leaf (applying only to MerkleDB), either 1000 or one; percentage of objects updated between anti-entropy runs, either 10% or 1%; and replication message loss, either 100% (making anti-entropy do all the work) or 10%. We evaluated 8 configurations for MerkleDB and 4 for DottedDB, each denoted with a sequence of letters, H or L, corresponding to each parameter choice, in the order they appear on the table. For example, MerkleDB *HHL* uses 1000 objects per

Table 3: Parameter choices used when evaluating Anti-Entropy. Objects per Leaf applies only to MerkleDB.

	Objects per Leaf	State Changed	Message Loss
High	1000	10%	100%
Low	1	1%	10%

leaf, performs anti-entropy when 10% of local storage has changed and loses 10% of replication messages when serving updates.

4.4.5.1 *Hit Ratio*

An anti-entropy mechanism should ideally only send objects to synchronize if they are really missing or outdated in the other node. We define *hit ratio* as the percentage of the objects sent via anti-entropy that were needed on the receiving side (i.e., not redundantly transmitted). For MerkleDB, we consider the number of key-hashes exchanged (relevant because in many cases object payloads are small, and the key-hash pair is a non-negligible part of the total size).

Figure 12 show the CDF of the hit ratio for all configurations. In most, MerkleDB has a hit ratio below 5%, with the exception of MerkleDB LLL with a hit ratio around 50%, which means that only in a most favorable scenario, the false positives for missing objects were relatively low. DottedDB exhibits a high hit ratio in all configurations, especially when syncing more frequently.

4.4.5.2 *Metadata Size*

MerkleDB uses only Merkle trees, but since peers do not replicate the same subset of keys, each node needs one Merkle tree per peer. Using consistent hashing and a replication factor of 3, each node needs 3 Merkle trees. The space used per Merkle tree is linear with the number of objects (key-hash lists in leaf nodes), plus the size of the tree itself, which is fixed upon bootstrap. The expected size of a single Merkle tree is:

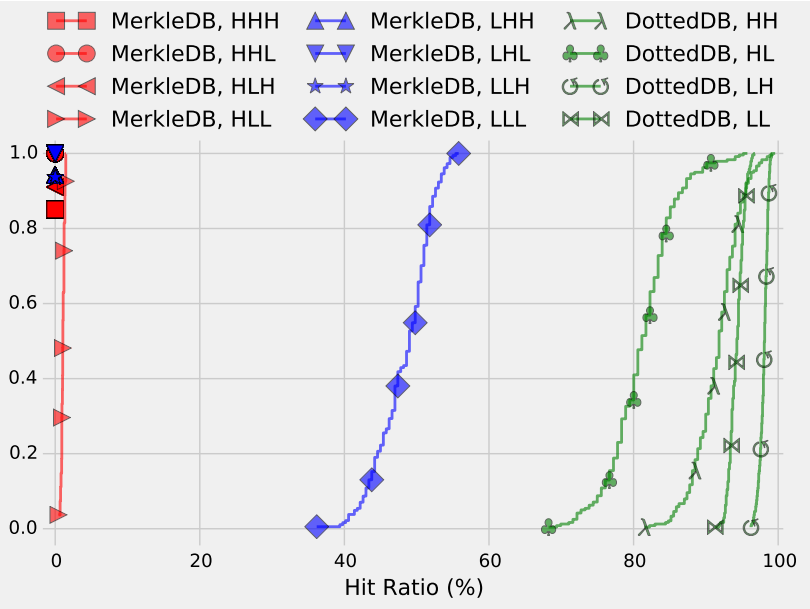


Figure 12: CDF of the hit ratio of the anti-entropy protocol.

$$Tree = (Hash + MD) \times \sum_{i=o}^{Height} BF^i$$

$$Lists = \frac{\#Keys \times RF}{\#Nodes} \times (Hash + MD + Key)$$

BF is the tree branching factor; *Height* is the tree height; *Hash* and *MD* are the sizes of hash and additional metadata per hash, respectively; *RF* is the replication factor; *#Keys* is the total number of keys in the database; *#Nodes* is the number of nodes; *Key* is the average key size. The total metadata per node in MerkleDB is given by $(Tree + Lists) \times 3$.

In DottedDB, metadata consists of the node clock, the dot-key map, the watermark and the non-stripped keys. In a quiescent state, the dot-key map and the non-stripped keys are both empty, and the node clock is simply a version vector. The watermark has constant size.

Figure 13 shows the node metadata over time for the same configurations as before. Since the entire key-set was preloaded, the number of objects is constant and thus the merkle tree is also constant. Having fewer keys per leaf (L** configurations) results in larger trees.

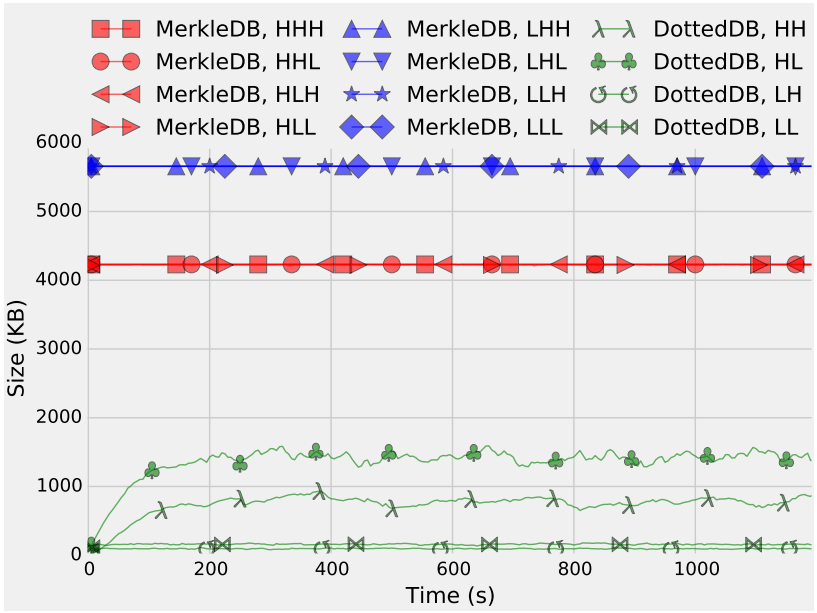


Figure 13: Metadata per node used by the anti-entropy protocol.

In DottedDB metadata is more dynamic, since certain structures grow and shrink depending on the system load and the sync interval. If the sync interval is small (configurations LH and LL) metadata is roughly constant and very small (< 10KB); but even in cases where syncs are infrequent, metadata remains mostly constant and much smaller than in merkle trees. While metadata in DottedDB depends mostly on the divergence, which can be kept small by frequent syncs, in MerkleDB it depends on the number of keys per node, which can be large.

4.4.5.3 *Network Usage*

MerkleDB sends each level of a Merkle tree in rounds, as necessary. If it reaches a leaf node, it sends all key-hashes in that leaf segment, to compare the hashes and see which objects are missing. DottedDB sends a combination of the node clock, watermark and missing objects.

Figure 14 shows the average network usage in runs for all configurations as before, segmented by *object data* (the key-value(s)), *object metadata* (object logical clock) and *sync metadata* (whatever was transferred by the protocol, excluding the object data and metadata). It can be seen that DottedDB is considerably more efficient in terms of network usage

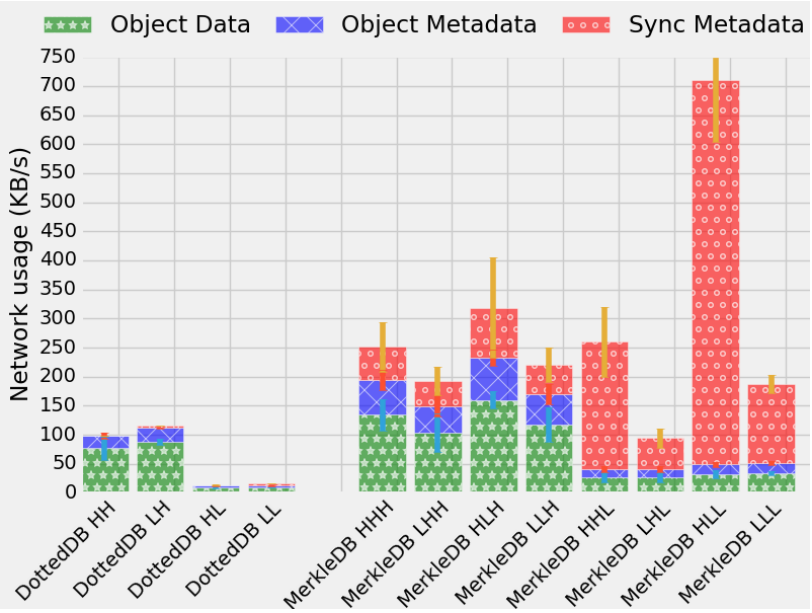


Figure 14: Average network traffic used by the anti-entropy protocol.

for all configurations. It is much more efficient in the most realistic *L scenarios, where network usage is much less than when all replication is through anti-entropy; comparatively, MerkleDB does not show such a decrease for **L scenarios when compared with **H ones, and sometimes the usage even increases, as from HLH to HLL.

Figure 15 shows the relative size between these three components (sync metadata, object metadata and object data), for a better assessment of their relative costs. Both systems have less anti-entropy overhead relative to object data+metadata when anti-entropy runs involve larger sets of objects in bulk, either because all replication is through anti-entropy (*H and **H configurations) or because a longer sync interval is used (H* and *H* configurations). In any case, for each corresponding scenario, the relative overhead in DottedDB is much less than in MerkleDB.

4.4.5.4 *Replication Latency*

Replication latency is the time between the initial write of a value in the coordinating node and the time the object is stored at another replica node. For 3 replicas per object each update request gives 2 different

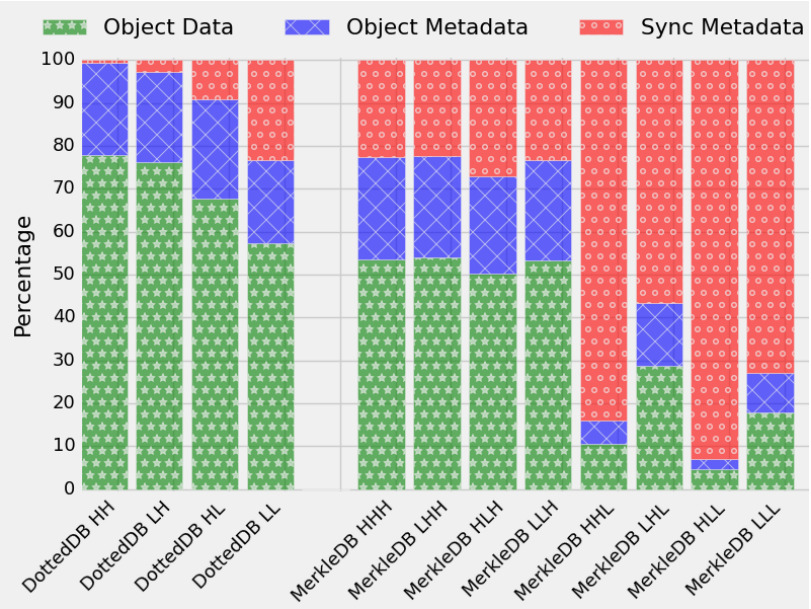


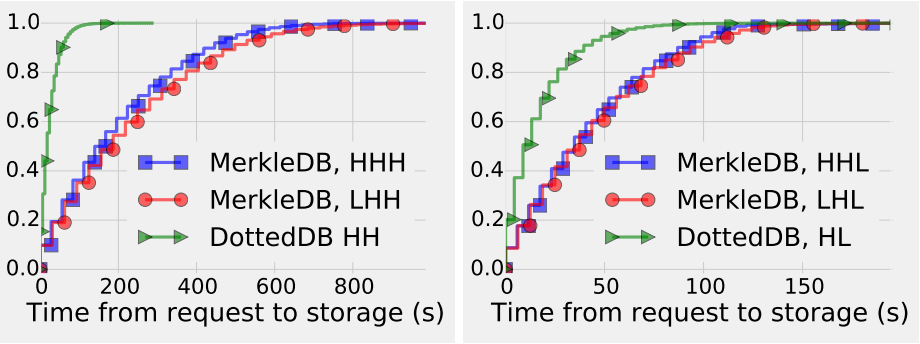
Figure 15: Relative proportions of object data, metadata and sync metadata exchanged in anti-entropy rounds.

times (the coordinator, always a replica node for the object, is excluded as its time would be 0). Figure 16 shows the CDFs for replication latency, where DottedDB is much faster at repairing data in every scenario. When replication message loss is 100% (replication exclusively by anti-entropy) and the sync interval is low (Figure 16c), DottedDB repairs 99% of missing data in less than 20 seconds, while MerkleDB takes 20 times more.

4.4.6 *Replication via Anti-Entropy*

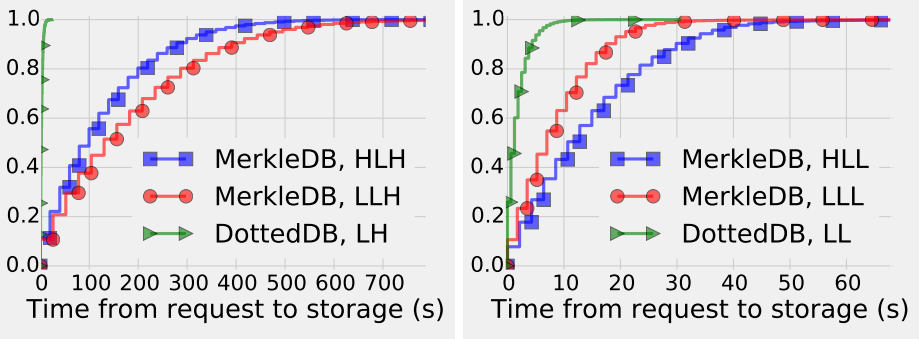
As an experiment, we disabled normal replication in DottedDB where nodes receive client writes and send the new object to other replica nodes. Instead, anti-entropy is run in fast intervals, being the only source of data replication.

We experimented with four different anti-entropy intervals: 1 ms, 10 ms, 100 ms and 1000 ms. Every other configuration is equal to the anti-entropy experiments run before.



(a) HH: *High* number of objects missing per anti-entropy (10%); *High* replication loss (100%).

(b) HL: *High* number of objects missing per anti-entropy (10%); *Low* replication loss (10%).



(c) LH: *Low* number of objects missing per anti-entropy (1%); *High* replication loss (100%).

(d) LL: *Low* number of objects missing per anti-entropy (1%); *Low* replication loss (10%).

Figure 16: CDFs of the replication latency: the time from the moment a node coordinates an update, until the object is stored at another replica.

4.4.6.1 *Replication Latency*

Figure 17 shows the CDF for the replication latency with varying anti-entropy intervals. As expected, data is replicated faster with more frequent anti-entropy rounds. In particular, 1 ms and 10 ms have very similar and fast replication latency profiles.

4.4.6.2 *Network Usage*

Figure 18 shows the relative usage of the network for the anti-entropy protocol. The metadata for the anti-entropy grows relative to other data with shorter intervals, since each round has less data to exchange. Also, the slower interval has the larger object metadata, which is probably

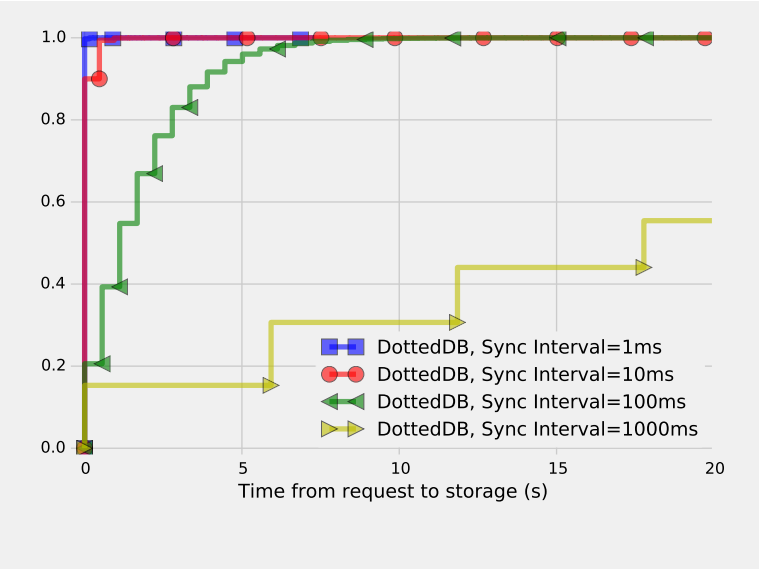


Figure 17: Replication Latency for various AE intervals.

a result of tying the node clock exchange for garbage collection with the anti-entropy rounds for data synchronization/replication. This suggests that it may be interesting to decouple the metadata exchange used to garbage collect data from the anti-entropy rounds, so that they can execute at different time intervals.

Figure 19 shows the network usage for the varying intervals tested. The fastest interval (1 ms) exchanges almost the same amount of metadata as the actual data it replicates, indicating that it may not be ideal for most scenarios, unless data freshness is the only consideration and the network is not overloaded. Using a 100 ms interval seems to yield few advantages when compared to 10 ms in terms of network usage, while using 10 ms has a much faster replication latency, as discussed previously. Doing anti-entropy rounds in 1 second intervals seems to be the less taxing in terms of network usage, which is expected since data is exchanged in larger batches, avoiding additional metadata and false positives.

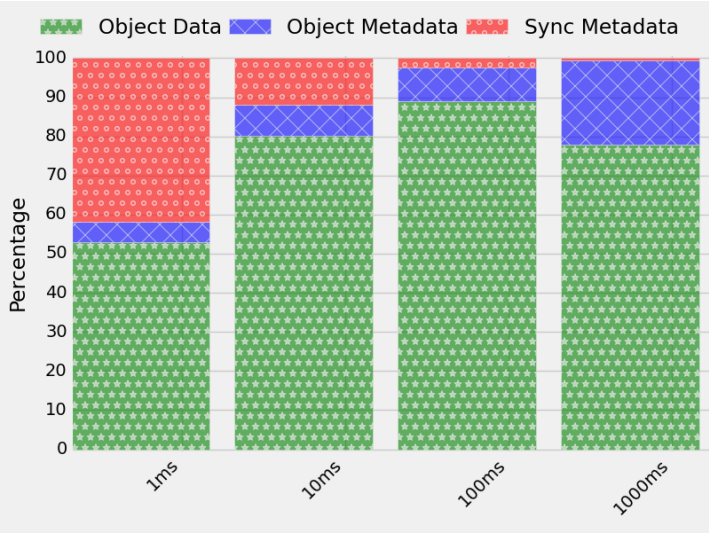


Figure 18: Relative proportions of object data, metadata and sync metadata exchanged in very-fast anti-entropy rounds.

4.4.7 *Client Request Latency*

Although neither system was particularly optimized for raw performance, we compared both in terms of client-perceived latency, using the same tests in the previous section.

Figure 20 shows the cumulative distribution for client request latency. DottedDB has a better latency curve all-around, except the tail latency with slower anti-entropy (H*). Through empirical observations, we know that these higher latencies are due to our implementation of anti-entropy, which exchanges objects in bulk, in combination with the fact that our nodes are single-threaded. This is detrimental to tail latencies when the number of objects to transfer is large (e.g. H*). MerkleDB uses Riak's merkle tree implementation, which exchanges key lists in separate leaf nodes independently, alleviating this problem. It's important to note that this is a limitation of our implementation and not an inherent limitation of our NDC framework.

Table 4 confirms the results by showing the average, the 95^{th} and the 99^{th} percentile for the same tests. The fast synchronization version of DottedDB (L*) is 33% faster on average than the fastest version of MerkleDB, and 86% faster on the 99^{th} percentile.

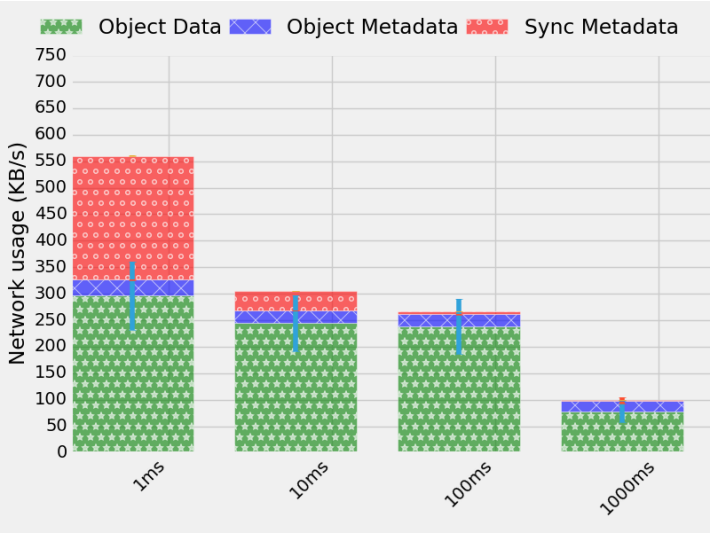


Figure 19: Average network traffic used in very-fast anti-entropy rounds.

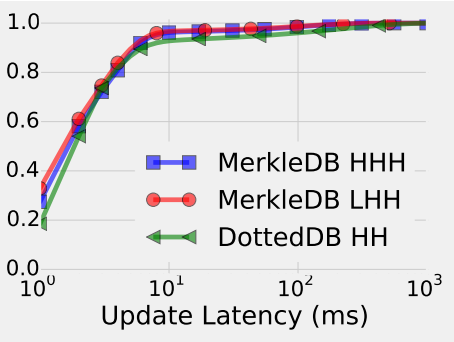
Table 4: The average, the 95^{th} and the 99^{th} latencies for client *Update* requests. The best result per line is in bold.

	DottedDB				**MerkleDB**							
	HH	**HL**	**LH**	**LL**	**HHH**	**HHL**	**HLH**	**HLL**	**LHH**	**LHL**	**LLH**	**LLL**
Avg.	19.8	16.4	**4.8**	5.1	7.0	9.3	6.7	8.1	6.4	8.9	7.0	8.0
95^{th}	64	39	8	**7**	8	11	8	10	8	11	8	10
99^{th}	460	394	**66**	79	137	173	128	155	123	163	143	149

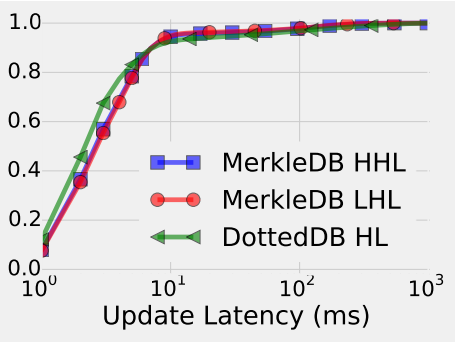
4.5 NODE-WIDE DOT-BASED CLOCKS WITHOUT FILL

There is an alternative approach to the NDC framework that never fills the object context with the node clock, in exchange for a slightly delayed garbage collection. We call this variant *NDC Without Fill* (NDC-NF) and the main idea is that if all current versions of the object are in all replica nodes, then the old object versions represented in the context have been removed everywhere. Thus, the context has served its purpose of eliminating older versions when exchanging and merging objects, and can be removed entirely.

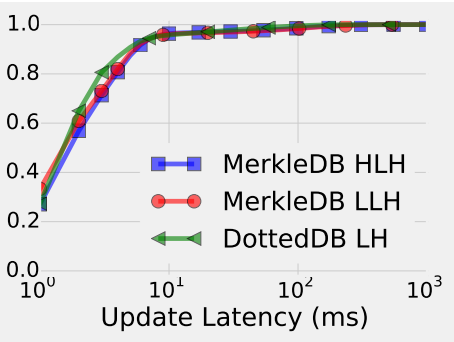
All data structures are the same as NDC, except for NSK that is no longer needed. Objects will be stripped when the dots of their current versions are removed from the DKM.



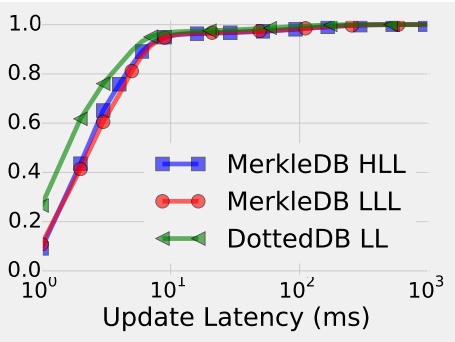
(a) HH: *High* number of objects missing per anti-entropy (10%); *High* replication loss (100%).



(b) HL: *High* number of objects missing per anti-entropy (10%); *Low* replication loss (10%).



(c) LH: *Low* number of objects missing per anti-entropy (1%); *High* replication loss (100%).



(d) LL: *Low* number of objects missing per anti-entropy (1%); *Low* replication loss (10%).

Figure 20: CDFs for the latency of client *Update* requests.

4.5.1 *Algorithms*

Algorithm 6 shows the client API for NDC-NF. The put and delete operations are the same, but the get operation must explicitly add the dots of the object versions to the causal context returned to the client. Without this, a stripped object with no context would send an empty context to the client, that if used in a subsequent update to the same key, would not dominate the versions previously read by the client.

Algorithm 7 defines the auxiliary operations for NDC-NF. Notably, the fill operation is removed, as it is no longer used in this version of NDC. Also, the merge operation remains the same.

The strip operation now only strips the causal context of an object if the version dots are known by all replica nodes of that key. When this

Algorithm 6: GET Operation at Node i with NDC-NF.

```
1  procedure GET(k : 𝕂, quorum[= 1] : ℕ):
2      O := ∅
3      parallel for j ∈ replica_nodes(k) do
4          O := O ∪ rpc(j, fetch, ⟨k⟩)
5      await(size(O) ⩾ quorum)
6      for o ∈ O do
7          (vers, cc) := merge((vers, cc), o)
       // add current dots to causal context for the client
8      return (ran(vers), cc ∪ dom(vers))
9
10 procedure PUT(...):
       // the same
11 procedure DELETE(...):
       // the same
```

happens, it means that the causal context has reached all replica nodes, and therefore removed all possible older versions present in the causal context.

The fetch operations now returns the object directly from storage, without the need to fill any causal context.

UPDATE The update operation now checks if all versions from the received object are already known by the node clock. If there is nothing new, then this object serves no purpose and it is ignored, returning the local object.

This new test in the update operation is not necessary in NDC but it is in NDC-NF, because nodes now are vulnerable to older objects that for some reason reach a node after the newer object is in all nodes; this should be extremely rare, but can happen with network reordering/delays.

The problem with older objects being replicated after removing the context of the current object is due to the assumption that every older object is now deleted. This is not relevant with NDC, since the context is always restored whenever the object is read from storage.

Versions of the incoming object that are already known by the local node are discarded before merging to the local object to avoid the reappearence of obsolete versions.

Algorithm 7: Auxiliary Operations at Node i with NDC-NF.

```
1  function fetch(k : 𝕂):
       // return the object as-is, without filling
2      return ST_i[k]
3
4  procedure store(k : 𝕂, o : Object):
5      for (j, c) ∈ dom(vers) do
6          NC_i[j] := NC_i[j] ∪ {c}
7          DKM_i[(j, c)] := k
8      ST_i[k] := o
9
10 function merge(...):
       // the same
11
12 function update(k : 𝕂, (vers, cc) : Object):
13     if dom(vers) ⊆ NC_i then return fetch(k)
14     vers := { (d, v) ∈ vers | d ∉ NC_i }
15     (vers, cc) := merge((vers, cc), fetch(k))
16     for (j, c) ∈ dom(vers) do cc[j] := max(cc[j], c)
17     store(k, (vers, cc))
18     return (vers, cc)
19
20 function strip((vers, cc) : Object):
       // test if all current dots are in all replica nodes
21     for (j, c) ∈ dom(vers) do
22         if min({ WM_i[p][j] | p ∈ peers(j) }) < c then
23             return (vers, cc)
24     return (vers, ∅)
25
26 function ~~fill(...)~~:
       // not needed anymore
```

STORE The store operation is now simplified to only update the node clock and the dot-key map and finally save the new object to storage. The process of stripping was delegated to another function that is called after anti-entropy rounds and the NSK is no longer needed.

4.5.1.1 *Background Processes*

Algorithm 8 shows the new definitions for NDC-NF background processes. The strip_causality process is removed, since the equivalent stripping process can be made by the gc_dkm procedure. When a dot is

Algorithm 8: Background Tasks at Node i with NDC-NF.

```
1  process strip_causality():
       // not needed anymore
2  process anti_entropy():
       // the same
3  procedure sync_clock(...):
       // the same
4
5  procedure sync_repair(p : 𝕀, NC : 𝕀 ↪ 𝒫(ℕ), O : 𝕂 × Object):
       // the same
6
7  procedure update_watermark(p : 𝕀, NC : 𝕀 ↪ 𝒫(ℕ)):
       // the same
8
9  procedure gc_dkm():
       // remove entries known by all peers
10     for ((j, c), k) ∈ DKM_i do
11         if min({ WM_i[p][j] | p ∈ peers(j) }) ⩾ c then
12             DKM_i := {(j, c)} ⩤ DKM_i
13             (vers, cc) := strip(ST_i[k])
14             if {val ∈ ran(vers) | val ≠ null} = ∅ ∧ cc = ∅ then
15                 ST_i := {k} ⩤ ST_i
16             else if cc = ∅ then
17                 ST_i[k] := (vers, ∅)
```

known by all replica nodes, it is the perfect time to check if the corresponding object can be stripped, because if that dot corresponds to the only version of the object, then that object is in all replica nodes and the context will be removed. This also the reason that the NSK data structure is no longer needed.

4.5.2 *NDC versus NDC-NF*

There are two disadvantages with the NDC-NF approach:

1. Object stripping is slightly slower than NDC, because the context is only removed when all versions are known to be in all replica nodes; in NDC, entries are removed independently when included in the node clock, which happens before that information is known by other peers;

2. The stripping depends on having information about *all* peer nodes, which makes the strip operation less fault-tolerant than the NDC version.

The main advantages of NDC-NF are the following:

1. The node clock can be safely garbage collected, because older entries from retired nodes are not used to fill objects;
2. The is no fill operation every time an object is read from storage, requiring less computation;
3. Objects in the local storage are all completely independent from the node clock, because they never require it for filling the causal context, which means that the node clock is not a single point of failure like in NDC and can be lost without affecting the context of the local objects;
4. The non-stripped keys are implicitly tracked by the dot-key map, and the stripping is only attempted when the chance of success is very high; thus, there is no need for an extra background process that constantly attempts to strip non-stripped keys.

4.6 DISCUSSION

Merkle Trees are very efficient digests when the changes they track exhibit spatial locality, such as when used for a hierarchical file-system. Not surprisingly, but apparently unnoticed, this efficiency goes away in systems that use consistent hashing to spread key allocation across nodes, as this destroys any locality patterns in the key space. Surprisingly, modern distributed key-value stores still adhere to this odd combination of techniques – maybe for the lack of an alternative. We have shown that consistent hashing and Merkle trees should not be used together; provided an alternative, based on tracking node-wide causality metadata; and demonstrated that it significantly improves the performance of currently used anti-entropy protocols.

An interesting outcome from the evaluation is the observation that, contrary to Merkle Trees, where there is a tradeoff between bandwidth overhead and repair latency, with the NDC framework using very frequent synchronizations is a win-win situation both in terms of bandwidth and latency. This opens up the possibility of discarding the traditional replication when serving client requests and leaving all replication to the anti-entropy mechanism.

In modern distributed key-value stores there has always been a tension among timestamp-based approaches, using last-writer-wins policies (e.g., Cassandra), and approaches that capture causality and represent concurrent updates for reconciliation (e.g., Riak). The former approach is often chosen due to its speed, simplicity and low metadata footprint, but this comes at the cost of arbitrary loss of updates under concurrency, given the lack of read-update-write transactions. The latter is more complex and incurs a much higher metadata cost. We have shown how to significantly reduce this cost, presenting a framework that minimizes the per-object metadata, without compromising accurate detection of concurrent updates. Our approach exhibits other two important benefits: allows correct distributed deletes with no need for permanent tombstones; works under node churn, while maintaining low metadata cost.

5

CAUSAL MULTI-VALUE CONSISTENCY

In distributed non strongly-consistent data stores today we have either classic single-value causal consistency, leading to lost updates, or multi-value memory API but with weaker consistency. In this chapter we define causal multi-value memory – causal memory having a multi-value API – under two variants (one more pure but unrealistic to implement, and another having per-location versioning). We discuss the feasibility of practical implementations of each variant, and compare each with other consistency models commonly used in distributed key-value stores.

5.1 CAUSAL MULTI-VALUE MEMORY

Let $P = \{p_i \mid i \in \mathbb{I}\}$ be the set of processes that access the memory. Let H_i the history of operations issued by process p_i, and $H = \{H_i \mid i \in \mathbb{I}\}$ the global history, for some arbitrary run. Operations can be either:

- $w_i^m(x, v)$: a write to location x of value v, being the m^{th} operation by p_i;
- $r_i^m(x, V)$: a read from location x, returning a set of values V, being the m^{th} operation by p_i.

We omit the process or sequence number if irrelevant for the context, or use a generic o for an operation which is either a read or a write. E.g., o_i^m is the m^{th} operation of process p_i, and $w(x, v)$ is some write of value v to location x.

In a *causal multi-value* (CMV) memory a read returns a set of values: an empty set if no write precedes it (i.e., for not yet initialized locations), and possibly more than one value if several concurrent writes precede it. For presentation simplicity, we consider each value written to be globally unique. (In an actual implementation, as we present below, if two writes use the same value, the memory must ensure that each value is uniquely tagged.)

A causality order of operations in H is determined by the program order in each process, and a *writes-into* order that associates a write operation with read operations that return the written value.

5.1.1 *Ordering*

5.1.1.1 *Process Order*

Process Order $\xrightarrow{p_i}$ is a binary relation on operations of the execution history H_i of process p_i, according to p_i local time:

$$o_i^m \xrightarrow{p_i} o_i^n \doteq m < n$$

The global process order relation $\xrightarrow{p}$ is the union of the process orders of all processes:

$$\xrightarrow{p} \doteq \bigcup_{i \in \mathbb{I}} \xrightarrow{p_i}$$

5.1.1.2 *Writes-Into Order*

Writes-into order is a binary relation induced by history H, relating writes and reads on the same location, defined by:

$$w(x, v) \xrightarrow{w} r(y, V) \doteq x = y \wedge v \in V$$

5.1.1.3 *Causality Order*

Causality Order ($\xrightarrow{c}$) is the relation formed by the transitive closure of the union of process order ($\xrightarrow{p}$) and writes-into order ($\xrightarrow{w}$):

$$\xrightarrow{c} \doteq \left(\xrightarrow{w} \cup \xrightarrow{p}\right)^{+}$$

i.e., $o \xrightarrow{c} o'$ is true only if at least one of the following properties holds:

1. $o \xrightarrow{p} o'$: o was executed before o' in the same process;
2. $o \xrightarrow{w} o'$: o' reads a set containing the value written by o;
3. $\exists o''. o \xrightarrow{c} o'' \xrightarrow{c} o'$: transitive closure.

If o and o' in H are two operations such that $o \not\xrightarrow{c} o'$ and $o' \not\xrightarrow{c} o$, then we say that they are *concurrent operations*.

5.1.1.4 *Causal Past*

The *causal past* of an operation o for some location x can be defined by the function $C^x(o)$, which returns the set of operations that accounts for all values written to x before o, according to the causality order:

$$C^x(o) \doteq \{\, w(x,\_) \mid w(x,\_) \xrightarrow{c} o \,\}$$

The entire causal past of an operation o is defined by the union of the causal past of o for all locations:

$$C(o) \doteq \bigcup_{x \in L} C^x(o)$$

5.1.2 *Causal Multi-Value Histories*

A history H is *Causal Multi-Value* (CMV) if and only if it respects the following three properties:

- *No cycles*: $\xrightarrow{c}$ induced by H is a strict partial order;
- *Read Some Write*:

$$r(x, V) \in H \Rightarrow \forall v \in V. w(x, v) \in H$$

- *Read Last Writes*: presented bellow in two variants.

The first condition forbids $\xrightarrow{c}$ cycles, preventing "reads from the future", or *out of thin air* values, which explain themselves in a causal loop. The second condition ensures that each value in the set returned by a read comes from some write to the corresponding location. The third condition dictates which values a read operation must return. There are two variants of this condition: a *global versioning* and a *per-location versioning*.

5.1.2.1 *Causal Multi-Value with Global Versioning Histories*

A history is Causal Multi-Value with *Global Versioning* (CMV-GV) if in addition to the first two conditions, it respects the following:

- *Read Last Writes with Global Versioning*:

$$o = r(x, V) \Rightarrow V = \{\, v \mid w(x, v) \in \max(C^x(o), \xrightarrow{c}) \,\} \tag{5}$$

It states that for a given operation o that reads from some location x, given the set of operations that write to x that are in the past of o according to the causality order, then o must return all values written by the maximal elements of that set. An element of a set is *maximal* under a given order, if it is not smaller than any other element, according to that order:

$$\max(S, <) \doteq \{\, e \in S \mid \nexists e' \in S.\, e < e' \,\}$$

In essence, the *Read Last Writes* dictates what values must be returned by a read, by defining what are the "current" and "old" values (or writes) at that point in history. Condition 5 provides global versioning, because it uses the global causality order $\xrightarrow{c}$ to compute the maximal elements of past writes. Since $\xrightarrow{c}$ is the transitive closure between process order and writes-into order, it relates all locations in different processes by a single read on some location by one process from another.

CAUSAL MULTI-VALUE WITH GLOBAL VERSIONING MEMORY A memory is CMV-GV if it only admits CMV-GV histories.

To define Causal Multi-Value with per-Location Versioning (CMV-LV) histories, we have to introduce a new per-location writes-into order and a per-location history order.

5.1.3 *Causal Multi-Value with per-Location Versioning Histories*

5.1.3.1 *Per-Location Writes-Into Order*

Writes-into order for a specific location x is a binary relation induced by history H, relating writes and reads on x, defined by:

$$w(y, v) \xrightarrow{w^x} r(z, V) \doteq x = y = z \wedge v \in V$$

The union of writes-into order for all locations in L is equal to the global writes-into order defined previously:

$$\bigcup_{x \in L} \xrightarrow{w^x} = \xrightarrow{w}$$

5.1.3.2 *Per-Location History Order*

History Order for some location x ($\xrightarrow{h^x}$) is the relation formed by the transitive closure of the union of process order ($\xrightarrow{p}$) and writes-into order for x ($\xrightarrow{w^x}$):

$$\xrightarrow{h^x} \doteq \left(\xrightarrow{w^x} \cup \xrightarrow{p} \right)^+$$

Since $\xrightarrow{w^x}$ is a subset of $\xrightarrow{w}$, each location's history order is also a subset of the causality order:

$$\xrightarrow{h^x} \subseteq \xrightarrow{c}$$

5.1.3.3 *Location History*

The *history* of an operation o for some location x can be defined by the function $H^x(o)$, which returns the set of operations in the global history that accounts for all values on x written before o, according to x's history order:

$$H^x(o) \doteq \{ w(x,\_) \mid w(x,\_) \xrightarrow{h^x} o \}$$

Note that, in the same way that the history order is a subset of the causality order, the history of an operation o for some location x, $H^x(o)$, is a subset of the causal past of that operation for that location $C^x(o)$.

5.1.3.4 *Causal Multi-Value with per-Location Versioning Histories*

A history is Causal Multi-Value with *per-Location Versioning* (CMV-LV) if in addition to the previously defined *No cycles* and *Read Some Write* properties, it respects the following:

- *Read Last Writes with per-Location Versioning*:

$$o = r(x, V) \Rightarrow V = \{ v \mid w(x,v) \in \max(C^x(o), \xrightarrow{h^x}) \} \tag{6}$$

The difference between global versioning (Condition 5) and per-location versioning (Condition 6) is that the former takes the maximal elements of the causal past of a read according to the *causality order*, while the latter takes the maximals of the same causal past, but according to the location's *history order*.

CAUSAL MULTI-VALUE WITH PER-LOCATION VERSIONING MEMORY
A memory is CMV-LV if it only admits CMV-LV histories.

5.2 FEASIBILITY OF CMVM FOR KEY-VALUE STORES

Consider a distributed key-value store that partially replicates a set of keys at multiple server nodes, which clients can read from and write to. Clients are the processes and keys are the locations of our model.

Clients do not communicate directly, only with the server. To implement CMVM, both the clients and the server should track metadata such as the *causality order* and the *history order*, depending on the versioning model used (global or per-location).

METADATA TRANSITIVITY Conceptually, a client starts with an empty metadata state, which is updated with every operation. A write operation updates the client state. That new state is also added to the key replica in the server, along with the new value. The key replica merges the new metadata state with its own state (which also starts empty).

A read operation uses both the client state and the key state to decide what to return to the client. The key state is returned and merged to the client state.

This metadata state lifecycle between the nodes in the server and clients provides the causality transitivity necessary between clients, since they do not communicate directly.

WRITE-ONLY PARTIAL ORDERS Each new write operation must be uniquely identifiable, while reads operations may not, since they do not produce new values or change the server state. A read operation only inflates the metadata state of a client session, in the same way that a read operation in the CMV memory model only introduced a new *writes-into* order between two subgraphs. Both the causality order and history order graphs can be accurately represented by write operations only, provided that all transitive relations between writes created by reads are present.

5.2.1 *Enforcing Read-Last-Write Property*

The values returned by a read operation must respect Property 5 or Property 6, for CMVM-GV or CMVM-LV, respectively.

5.2.1.1 *Global Versioning*

The metadata necessary to enforce Property 5 is only the causality order. This order serves two purposes: (a) it provides the causal past $C(o)$; (b) it provides the partial order necessary to compute the maximal elements of the causal past. Therefore, each client and each key replica in a server node should maintain the causality order, necessary to enforce CMVM-GV.

Each write operation is added to the causality order as being in the future of every previous operation (representing the *process order*). The resulting causality graph is merged with the key's causality graph at the node performing the write operation. A read operation sends the client causality graph to the server, which is merged with the key causality graph to compute what values (corresponding to individual write operations in the graph) must be returned to the client. The merged causality graph is saved as the current client causality graph (representing the *writes-into order*).

5.2.1.2 *Per-Location Versioning*

To enforce Property 6, the metadata should include the causality order and the history order per key. The former is used solely to give the causal past of the read operation $C(o)$, while the latter is used to compute the maximals of that causal past, according to the key's history order.

The process for updating the causality order is the same as in the Global Versioning. However, the history order is tracked separately per-key. Similarly to the causality graph, a new write operation is added to the history graph of that key, as being in the future of every other write in the graph. Both graphs are sent to the server in a write operation, to be merged and saved in that key replica.

A read operation sends both graphs to the server, which the key merges with its own graphs to compute: (a) the values in the causal past using the causality graph, and (b) the maximals of the writes in the causal past to return to the client. In addition to receiving the set

of values, the client receives and saves the merged causality and history graph.

5.2.2 *Garbage Collecting Metadata*

As discussed, the key metadata has two roles in the process of computing the maximal values to return in a read operation. The first role is to compute the causal past of that key, and the second role is to take that result and apply a partial order over it to compute the maximal elements. In other words, the causal past demands the *visibility* of a set of writes on that key, and the partial order selects the obsolete writes for this key replica in that set, so that only non-obsolete writes are returned to the client.

CAUSAL PAST The causal past represents the potential write operations to be seen by a read operation. But if a write in the causal past is present in all replicas of the corresponding key, it is already guaranteed that that write will be considered and potentially returned by a read operation. Thus, each write in a causal past can be individually removed if it is present in all replicas of the corresponding key:

$$\mathsf{GC}(\mathsf{C}(o)) = \{\, o'(k,\_) \in \mathsf{C}(o) \mid o' \text{ is not in all replicas of } k \,\}$$

PARTIAL ORDER FOR MAXIMALS The partial order (either the causality order or the history order) is used to determine the maximal elements of the causal past. In others words, it is used to guarantee that no *old* value is returned alongside a *newer* value. If the maximals for a given replica key are present in all other replicas, then all non-maximals (older values) of that key have been removed in the server. Thus, non-maximal elements are no longer necessary to be maintained in the partial order. However, the maximals of all keys in the partial order are always necessary to identify the current values, which must be obsoleted if for example a client reads a key and then writes to another key in the partial order.

For example, consider a server with CMVM-GV semantics containing a replica of the key x, which stores the causality order with the maximals $w(x,4)$ and $w(y,3)$. If a client reads that replica of x and then writes $w(y,4)$, that new write should be in the future of the maximals of y, which were returned by the previous read on x. Without storing the maximals for all keys in the causality order in *every* replica of *every* key, the client would not have known that $w(y,3)$ was the current maximal for y, and the $w(y,4)$ would then be considered concurrent with it.

5.2.3 *CMVM-GV vs CMVM-LV after GC*

Accounting for the garbage collection described previously, it becomes clear that CMVM-LV requires *much less* metadata to be stored in the server (and in the clients, if they also perform GC). While the causal past is equally garbage collected by both, CMVM-LV only stores, per key replica, maximals of the history order for that key, while CMVM-GV stores, per key replica, the maximals for *all* the keys in the causality order.

Thus, in a quiescent system where every possible metadata was garbage collected, implementing local versioning only requires keys to identify their maximal writes, corresponding to their current values. The space complexity for this case is $O(1)$, since typically most keys only have one value.

Implementing global versioning requires keys to identify the maximals of all keys in their causal past, which tends to grow, as keys are increasingly related by new client read operations. The space complexity in this case is $O(K)$, where K is the number of keys in the causal past (with enough operations, K contains all keys).

In conclusion, CMVM-LV is the scalable variant to implement CMVM semantics in a distributed data store. From a client's perspective, the only difference is that CMVM-LV requires clients to read a key before updating it, while in CMVM-GV all keys in the client causal past are

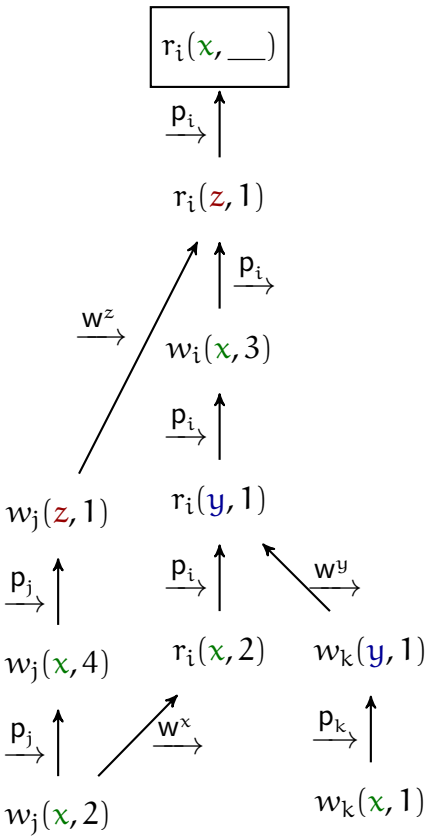


Figure 21: The process order and the writes-into order for all operations from processes i, j and k, on keys x, y and z.

updated implicitly by new writes, without necessarily having read them before.

5.2.4 *Consistency Models Comparison*

To aid in our comparison, consider the example illustrated in Figure 21. We have there processes (i, j, k) making read and write operations to three keys: x, y, z. The result of the last read by i on x is defined on Table 5, for various types of consistency.

5.2.4.1 *CMVM with Global Versioning*

The causal past of the last read by i on x is:

Table 5: The result of the last read by process i on key x, as in Figure 21, using different consistency models.

Model	**Result for the last read** $r_i(x, V)$
CMVM-GV	$V = \{3, 4\}$
CMVM-LV	$V = \{1, 3, 4\}$
CM	$V \in \{3, 4\}$
EC w/ per-key CC	$V \in \{ s \in \mathcal{P}(1, 2, 3, 4) \mid \{2, 3\} \not\subseteq s \wedge \{2, 4\} \not\subseteq s \}$
EC w/o per-key CC	$V \in \{ \bot, 1, 2, 3, 4 \}$

$$C^x(r(x, V)) = \{w(x, 1), w(x, 2), w(x, 3), w(x, 4)\}$$

According to the causality order, value 3 is in the future of value 1, and both values 3 and 4 are in the future of value 2:

$$w(x, 1) \xrightarrow{c} w(x, 3) \wedge w(x, 2) \xrightarrow{c} w(x, 3) \wedge w(x, 2) \xrightarrow{c} w(x, 4)$$

Therefore, values 3 and 4 are the maximals of the causal past, which are returned to the client. Notice that process i never read $w(x, 1)$ directly, but because it read $w(y, 1)$ which had $w(x, 1)$ in its causal past, the subsequent write $w(x, 3)$ introduced an order between the two writes, which made $w(x, 1)$ a past version of $w(x, 3)$.

5.2.4.2 *CMVM with Local Versioning*

The causal past of the last read by i on x is the same as in global versioning:

$$C^x(r(x, V)) = \{w(x, 1), w(x, 2), w(x, 3), w(x, 4)\}$$

However, according to the x's history order (see Figure 22), the only relations between writes on x are:

$$w(x, 2) \xrightarrow{h^x} w(x, 3) \wedge w(x, 2) \xrightarrow{h^x} w(x, 4)$$

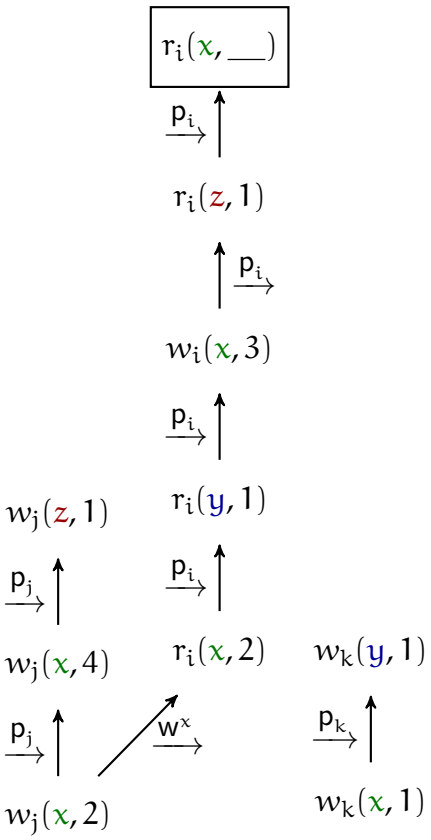


Figure 22: The process order and x's writes-into order for all operations from processes i, j and k, on keys x, y and z.

Therefore, the maximals of the causal past according to x's history order are the values 1, 3 and 4. Notice that the difference between GV and LV is that LV does not considers the writes-into orders for other keys to decide the maximals of a given key. In other words, with CMVM-LV, the client only introduces order between writes to the same key if it reads those writes directly. In this example, process i never read $w(x, 1)$ directly, which means that its write $w(x, 3)$ is not in the future of $w(x, 1)$ according to x's history order:

$$w(x, 1) \xrightarrow{c} w(x, 3) \wedge w(x, 1) \not\xrightarrow{h^x} w(x, 3)$$

5.2.4.3 *Causal Memory*

Causally consistent distributed data stores implement Causal Memory (CM) [1]. They capture the causality order like CMVM, but do not perform global or per-location versioning. They instead arbitrate a "winning" write if there is more than one causal maximal. This can be imagined as having an arbitration total order for concurrent writes. The arbitration can be made with Lamport Clocks [34] or wall-clock timestamps, for example. With timestamps, the latest write according to the timestamp wins, also called the *last-write-wins* rule. Thus, with CM semantics writes can be *silently lost*.

In our example, the result of the last read could be either value 3 or 4, but not both. Assuming that $w(x, 4)$ had a higher timestamp than $w(x, 3)$, value 3 written by i would be essentially overwritten by j, without j ever knowing of its existence.

5.2.4.4 *Eventual Consistency with Per-Key Causal Consistency*

Some distributed key-value stores that offer eventually consistency use per-key causal consistency, which can be implemented with logical clocks like Dotted Version Vectors [4]. It is analogous to the local versioning made by CMVM-LV, without the causality order.

In this example, the only thing that can be guaranteed is that the set of values returned by the last read cannot have versions that are not concurrent. I.e., if either value 3 or value 4 is returned by the read, value 2 cannot be returned, because the logical clock represents a partial order that says that $w(x, 2)$ is in the past of those two writes. Any other combination is possible, including an empty set, since the replica of x can be in any state, including not having seen any write for x yet.

5.2.4.5 *Eventual Consistency without Per-Key Causal Consistency*

Distributed key-value stores that do not implement per-key causal consistency, arbitrate what write to return in a read, typically by choosing the value with the highest timestamp. Therefore, any individual value can possibly be returned, including a "not found", since the replica of x can be in any state, including not having seen any write for x yet.

5.2.5 *Discussion*

5.2.5.1 *Causal vs Eventual*

One observation that can be made regarding the previous example is that eventually consistent systems provide very little guarantees to clients. For example, a client can update a key and later read an older version of that key. Another example is a client reading a version of a key and later reading a past version of that key. Causal models (CM/CMVM) on the other hand, provide *Session Guarantees* [56] to clients, which prohibit these anomalies and others. They provide the following four properties:

- *Read Your Writes*: read operations reflect previous writes;
- *Monotonic Reads*: successive reads reflect a non-decreasing set of writes;
- *Monotonic Writes*: writes are propagated after writes that logically precede them;
- *Writes Follows Reads*: writes are propagated after reads on which they depend.

5.2.5.2 *Causal Models*

There are many causally consistent systems that have scalable implementations [17, 18, 36, 37]. There are two main reasons for this. One is the fact that CM systems don't need to store maximal elements of a partial order, because they arbitrate the winner by some deterministic algorithm. As previously mentioned, this ultimately can cause silent data loss.

Another way to make CM scalable is by blocking write operations, until they are certain that the client causal past is present in all replicas. This is a pessimist approach to enforce causality, slowing writes until remote dependency checks are made. They however avoid having to store the dependencies at the server at any time. This approach could also be used with CMVM, but in our view the latency penalty is not

worthwhile, given that CMVM-LV has a small metadata footprint. Also, it goes against the writes-always-available Dynamo spirit, given that partitions occur, even inside single datacenters.

In conclusion, CMVM-LV can be seen as the best of both worlds: it has the metadata scalability of CM systems, enforces causal relationships, as CMVM-GV, and supports concurrent writes while avoiding data loss.

6

A DISTRIBUTED CAUSAL MULTI-VALUE DATA STORE

In this chapter we provide a distributed implementation in the form of a key-value store supporting partial replication with partial overlapping. We introduce a basic reference design for such system and then proceed to optimize the algorithms, such that it can scale with increasing data and nodes. We also discuss some alternative design choices that can be made when implementing a CMVM-LV data store.

6.1 BASIC REFERENCE DESIGN

We will now define an implementation of a distributed key-value store that provides CMVM-LV semantics without the complexity of garbage collection. This is a simpler implementation to explain initially, but would not be practical since metadata tends to grow unbounded. The implementation with garbage collection will be defined in a next section.

6.1.1 *System Model*

We consider a partially replicated master-less key-value store, where every node in the system is available to serve client requests for keys that it replicates. Nodes are totally independent from each other and have their own local storage. Nodes communicate via message passing, which can be delayed, reordered and lost by the network.

A node that receives a client request for a key that is not replicated locally, forwards the request to one of the replica nodes of that key. The replica node that receives the request is called the *coordinator* node. The coordinator may or may not need to contact other nodes to satisfy the client request. Nodes that replicate a common (not empty) set of keys are called *peer* nodes.

Clients can create, read, update and delete single objects, which are identified by a unique key. An object can have one or more values, as defined by a causal multi-value memory. Deletes write are associated with special *null* values.

6.1.2 *Causality Metadata*

As discussed in the previous chapter, to support CMVM-LV, both clients and the server should keep track of some metadata to enforce the causality order and local key versioning. We will define how to uniquely identify writes, how to represent the causal past and the history order.

6.1.2.1 *Dot: Unique Write ID*

Each client write associates a value with a key (deletes are treated as writes in our system). To uniquely identify this value to this key, the coordinator node upon receiving a new write request, associates the value with a new globally unique identifier, which we call *dot*: a pair made up of an identifier of the coordinator node and a unique counter for that coordinator. Its definition is as follows:

$$\mathbb{D} \doteq \mathbb{I} \times \mathbb{N}$$

We call each (dot,value) pair a *version* of an object or key.

6.1.2.2 *Causal Past*

As already discussed, the causal past is needed to enforce causality order. Both clients and objects associated with keys track the causal past, which flows from clients to server with writes, and from server to

clients with reads. Since each write is associated with a new dot, the causal past can be a collection of dots. Since it is useful to know which key some dot refers to, the causal past definition also maps key to dots representing writes made to that key:

$$\mathbb{C} \doteq \mathbb{K} \hookrightarrow \mathcal{P}(\mathbb{D})$$

Because the causal past of a dot (representing a write operation) is conceptually a set of dots, the causal past of any replica object o is the union of the causal pasts of all writes W_o made to that object:

$$C(o) \doteq \bigcup_{w \in W_o} C(w)$$

The causal past of a client c at any given time is the union of causal past for all operations O_c that it executed:

$$C(c) = \bigcup_{o \in O_c} C(o)$$

We can also call the causal past the *dependencies* of the object or the client, a term commonly used in causally consistency systems.

6.1.2.3 *Per-Key History*

Since we are implementing CMVM with per-Location Versioning, the system also has to capture the per-key history order. However, what is actually needed by CMVM-LV is to distinguish maximals from non-maximals elements of the causal past according to the per-key history order. To satisfy a read request, the coordinator node takes the union of the client causal past and the object causal past, and returns the values whose dots are from that key and are not in the history of that object. The current maximals read are joined with the key history by the client, for future writes.

We can represent the history order of the key as a collection of dots, representing all *non-maximals* of the history order. It is defined as follows:

$$\mathbb{H} \doteq \mathcal{P}(\mathbb{D})$$

For simplicity we are representing an object's history as a set of dots, but it could be represented as a compact version vector, similar to other systems such as DottedDB [22].

The history of an object o of key x is equal to the union of the *location histories* for x of all writes W_o made to that object:

$$\mathsf{H}^x(o) = \bigcup_{w \in W_o} \mathsf{H}^x(w)$$

6.1.3 *Client-Server API*

The servers allows clients to read, write or delete values associated with keys. Deleting is a particular write case with a special $\mathtt{null}$ value. The two public operations have the following signature type:

$$\begin{aligned} &\mathsf{Get}(\mathbb{K} \times \mathbb{C}) && \to (\mathbb{D} \hookrightarrow \mathbb{V}) \times \mathbb{C} \times \mathbb{H} \\ &\mathsf{Put}(\mathbb{K} \times \mathbb{V} \times \mathbb{C} \times \mathbb{H}) && \to \mathbb{D} \end{aligned}$$

A write operation sends a new value for the key and sends its current dependencies and the history for that key. A delete operation can be accomplished by mapping it to a write operation with a $\mathtt{null}$ value, which reads can later filter out. Write operations create a new version of that key, which has associated a new dot created by the coordinator node. That dot is returned to the client to add it to the dependencies and to the history of that key.

The read operation sends to the server the current dependencies of the client. The history of the key in the client is not sent, because it is information that objects in the server already have. As for the result of a read request, the coordinator returns the set of causally concurrent versions, its dependencies and its history.

6.1.4 *Client Library*

Given the necessity of handling causality metadata at the client side, clients should use a lightweight client library to interface with the data store. This library uses the client-server API discussed before, but takes care of managing the necessary metadata between operations to maintain CMVM-LV semantics.

Clients can run multiple independent "processes", encapsulating each state in a *session* object. The metadata necessary for a session is: (a) the session dependencies; (b) the history per key used in operations in the session. A session object can be defined as follows:

$$\mathsf{Session} \doteq \mathbb{C} \times (\mathbb{K} \hookrightarrow \mathbb{H})$$

As previously stated, the causal past of a session is equal to the union of the causal past of all operations in that session. The session also stores the most recent key history for each key read or written in the session.

A session is initialized with an empty set of dependencies and an empty map of key histories: $(\emptyset, \emptyset)$. Sessions can even be merged to create a new session object that has both causal pasts and histories. The operations available in the client library are the following:

$$\begin{aligned}
\mathsf{Read}(\mathsf{Session} \times \mathbb{K}) &\to \mathsf{Session} \times \mathcal{P}(\mathbb{V}) \\
\mathsf{Write}(\mathsf{Session} \times \mathbb{K} \times \mathbb{V}) &\to \mathsf{Session} \\
\mathsf{Delete}(\mathsf{Session} \times \mathbb{K}) &\to \mathsf{Session} \\
\mathsf{Merge}(\mathsf{Session} \times \mathsf{Session}) &\to \mathsf{Session}
\end{aligned}$$

Although we defined an explicit session object to be maintained by the client, it could be abstracted away in the client library by a mechanism (e.g., using thread-local storage), that automatically manages the session object lifecycle.

Algorithm 9: Client Library Operations.

```
1  procedure Read((C, H) : Session, k : 𝕂):
2  |  (V, C′, H′) := rpc(k, Get, ⟨k, C[k]⟩)
3  |  C := C ⊔ C′
4  |  H[k] := H[k] ∪ H′ ∪ dom(V)
5  |  return ((C, H), {v ∈ ran(V) | v ≠ null})
6
7  procedure Write((C, H) : Session, k : 𝕂, v : 𝕍):
8  |  d := rpc(k, Put, ⟨k, v, C, H[k]⟩)
9  |  C[k] := {d} ∪ C[k]
10 |  H[k] := {d} ∪ H[k]
11 |  return (C, H)
12
13 procedure Delete((C, H) : Session, k : 𝕂):
14 |  return Write((C, H), k, null)
15
16 procedure Merge((C1, H1) : Session, (C2, H2) : Session):
17 |  return (C1 ⊔ C1, H1 ⊔ H2)
```

6.1.5 *Client Library Algorithms*

Algorithm 9 defines the operations that the client library exposes to the client. All operations explicitly receive a session and return the updated session.

AUXILIARY FUNCTIONS We assume the definition of the functions rpc / async_rpc(target, fun, args). rpc executes synchronously, while async_rpc executes asynchronously. Both receive a target, a procedure to execute and a list of arguments. The target can be: 1) a *key*: it routes the message to one of the replica nodes of that key; 2) a *node id*: it routes the message to that node; 3) keyword *random*: it routes the message to a random node of the data store. Also we assume the definition of the function replicas(k), which returns the set of replica nodes for the key k.

READ The read operation takes a session object and a key, and calls the Get operation on some replica node of the key, sending the depen-

dencies corresponding to that key. The server returns an *object* associated with that key.

An object has a set of versions — dot-value pairs — which represents the maximals of this object. In addition to the current versions, an object stores its dependencies and history. A special value $\mathtt{null}$ is used in versions created by delete operations. An object is defined as follows:

$$\mathsf{Object} \doteq (\mathbb{D} \hookrightarrow \mathbb{V}) \times \mathbb{C} \times \mathbb{H}$$

The maximal versions returned by the server respects both client and the object causality and history order. The dependencies and history of the object are added to the session metadata, and the maximal values are returned (except for $\mathtt{null}$ values from delete operations).

WRITE/DELETE A write operation receives the session object, the key and the new value. It calls the Put operation on some replica node of the key, sending all session dependencies and the history of that key. The server returns the dot of the new value, to be added to the session dependencies and to the key history. The Delete operation is simply mapped to the corresponding Write operation with a $\mathtt{null}$ value.

MERGING SESSIONS Merging two sessions can be accomplished by the point-wise union of the dependencies and the histories from each session.

6.1.6 *Server Node State*

In addition to a unique identifier and a local object storage, each node will also maintain a *node history* and a map to allow finding which key correspond to a given dot.

6.1.6.1 *Storage*

Each node has its own durable object storage. The storage is defined as a map from keys to objects, as follows:

$$\mathsf{ST}_i : \mathbb{K} \hookrightarrow \mathsf{Object}$$

6.1.6.2 *Node History*

The node history is a data structure that contains the union of the histories of all object stored locally, in addition to all the current versions. The node history is used to generate new unique dots, and is also used by the anti-entropy protocol to synchronize nodes by comparing node histories. The history of some node i is defined as follows:

$$\mathsf{NH}_i : \mathbb{H}$$

The node history for some node i must respect Invariant 1, which states that at any time, the node history must be *extrinsic* (see Definition 4.2.1) to the history and the current version dots of all objects stored locally. All other events contained in the node history that are not part of the right hand side of the invariant are events from other keys not stored by the node.

Node History Invariant 1.

$$\mathsf{NH}_i \text{ is extrinsic to } \bigcup \{ \mathsf{H} \cup \mathsf{dom}(\mathsf{V}) \mid (\mathsf{V}, \mathsf{C}, \mathsf{H}) \in \mathsf{ran}(\mathsf{ST}_i) \}$$

For simplicity, the node history is represented here as a set of dots, but more compact data structures are available, such as the combination of a version vector for contiguous dots and bitmaps for sparse dots (similar to [22]).

6.1.6.3 *Dot-Key Map*

Each node also stores a mapping from dots of objects stored locally to the corresponding keys, so that the anti-entropy protocol can know which objects to read and send to another node that is missing those dots. We call this data structure the *Dot-Key Map* (DKM), and it is defined as follows:

$$\mathsf{DKM}_i : \mathbb{D} \hookrightarrow \mathbb{K}$$

Algorithm 10: Read request at Node i.

```
1 procedure Get(k : 𝕂, C^k : 𝒫(𝔻)):
2     (V, C, H) := fetch(k)
3     while (C^k ∪ C[k]) ⊈ H do
4         o := rpc(k, fetch, ⟨k⟩)
5         (V, C, H) := update(k, o)
6     return (V, C, H)
```

STATE INITIALIZATION A node server i is initialized with an empty node history, an empty storage and an empty dot-key map:

$$\mathsf{Init}(i) = (\mathsf{NH}_i := \emptyset, \mathsf{DKM}_i := \emptyset, \mathsf{ST}_i := \emptyset)$$

6.1.7 *Server Algorithms*

Each node will implement the client-server API in addition to a background *anti-entropy* process to exchange metadata between nodes and provide data convergence.

6.1.7.1 *Reads*

Algorithm 10 shows the definition of the Get operation at some node i. After reading the local object, the object history is used to check if the object and the client dependencies are locally met. Only dependencies of the key being read are relevant to the result of the operation.

If not all dependencies are locally met, then other replicas of this key are fetched, until all dependencies are met. Here, for simplicity, we fetch from a random replica node, but we could use the node id inside the missing dependencies to decide what replica node to contact first. Fetched objects update the current local object (if they have new versions). After having all the dependencies, the current object is returned to the client.

There is a possibility of getting in an infinite cycle of dependency fetching, since newer objects can have still have dependencies not locally met. This could only happen if other clients were continuously updating the same key. It is unlikely that long cycles of fetching would

Algorithm 11: Write Request at Node i.

```
1 procedure Put(k : 𝕂, v : 𝕍, C : ℂ, H : 𝕀):
2     c := max({ c | (j, c) ∈ NH_i ∧ j = i })
3     d := (i, c + 1)
4     C[k] := {d} ∪ C[k]
5     o := update(k, ({(d, v)}, C, H))
6     for j ∈ replicas(k) do
7         async_rpc(j, update, ⟨k, o⟩)
8     return d
```

occur, but if that was a concern, nodes could temporarily cache obsoleted versions to satisfy read dependencies without introducing newer dependencies from concurrent client updates.

6.1.7.2 *Writes*

Algorithm 11 defines the Put operation. Using the node history, a new unique dot is generated to be associated with the new value. The new dot is added to the dependencies sent by the client, which, together with client history and the new version, forms a new temporary object. This object is then merged with the local stored object and the result is stored and asynchronously replicated. Finally, the new dot is returned to the client.

6.1.7.3 *Anti-Entropy Protocol*

Given the assumption that messages between nodes can be lost, nodes can become out-of-sync and have unbounded stale data. To address this, nodes periodically run an *anti-entropy* protocol to repair missing/stale data across nodes. The main idea is to compute the set difference of the node histories from each pair of peers. This gives the exact set of dots that are missing from one node that the other can provide.

Algorithm 12 shows the definition of the anti-entropy process, which continuously runs in the background on each node i. In fixed intervals, it starts an anti-entropy round with a random peer node j, sending its history to compute the missing data.

Algorithm 12: Anti-Entropy Protocol at Node i.

```
1  process AntiEntropy():
2      loop forever
3          j := random(peers(i))
4          async_rpc(j, SyncHistory, ⟨i, NH_i⟩)
5          sleep(Δ)
6
7  function SyncHistory(p : 𝕀, H : ℍ):
8      K := { k | (d, k) ∈ DKM_i ∧ d ∉ H ∧ p ∈ replicas(k) }
9      O := { (k, fetch(k)) | k ∈ K }
10     H^i := { (j, c) ∈ NH_i | j = i }
11     async_rpc(p, SyncRepair, ⟨i, H^i, O⟩)
12
13 procedure SyncRepair(p : 𝕀, H^p : ℍ, O : 𝕂 ↪ Object):
14     for (k, o) ∈ O do
15         update(k, o)
16     NH_i := NH_i ∪ H^p
```

Node j collects all keys whose dots in the dot-key map are not present in node i history, while ignoring keys that are not replicated by i. It then fetches all local objects corresponding to those keys and sends them to i, along with all dots in j's history that were generated locally.

Finally, node i receives the missing objects and updates its local storage accordingly. All dots generated by j are added to i's node history, to close any gap from dots of objects generated at j but not replicated by i (necessary for a compact representation like a version vector).

Although node j sends to node i all of its local objects missing from i, j's entire history cannot be merged with i's history, because of possible partial overlapping in replication: j's history can contain some dots by a third node k whose keys are not replicated by j, but are replicated by i; in that case, merging the j's entire history into i's history, would indicate that i received those writes from k; however, that would be false because j does not replicate those keys and thus could not have sent them to i.

Algorithm 13: Auxiliary Procedures at Node i.

```
1  function fetch(k : 𝕂):
2      if k ∈ dom(ST_i) then
3          return ST_i[k]
4      else
5          return (∅, ∅, ∅)
6
7  procedure store(k : 𝕂, (V, C, H) : Object):
8      NH_i := NH_i ∪ H
9      for d ∈ dom(V) do
10         NH_i := NH_i ∪ {d}
11         DKM_i[d] := k
12     ST_i[k] := (V, C, H)
13
14 function merge((V_1, C_1, H_1) : Object, (V_2, C_2, H_2) : Object):
15     V_1 := { (d, _) ∈ V_1 | d ∉ H_2 }
16     V_2 := { (d, _) ∈ V_2 | d ∉ H_1 }
17     return (V_1 ⊔ V_2, C_1 ⊔ C_2, H_1 ∪ H_2)
18
19 procedure update(k : 𝕂, (V, C, H) : Object):
20     o := fetch(k)
21     if dom(V) ⊈ NH_i then
22         o := merge((V, C, H), o)
23         store(k, o)
24     return o
```

6.1.7.4 *Local Operations*

In addition to the client-server API and the anti-entropy protocol, each node also defines the following local operations (see Algorithm 13):

- fetch: if the key has an object in storage, that object is returned; otherwise, an empty object is returned;
- store: receives an object to save to storage; it also adds the dots of the object versions to the node history and to the dot-key map (for future anti-entropy rounds); the object history is also added to the node's history, to maintain the *Node History Invariant*;
- merge: receives two objects to be merged into a new one. The dependencies and histories are simply merged together. Versions

whose dots are present in the other object's history are removed, since it makes them obsolete;

- update: receives an object to update the local storage. It first checks if the received object has in fact new information. If not, the node has already seen all versions of that object, and thus the current local object is returned; otherwise, the new object is merged with the local object, stored locally and then returned.

6.2 OPTIMIZED DESIGN WITH GARBAGE COLLECTION

We defined a framework for a distributed key-value store that provides causal multi-value consistency with local versioning. The downside of this implementation is that metadata is always increasing with new operations. Even if sessions are short-lived and their metadata is temporary, the server metadata keeps increasing and more dependencies means larger client-server messages.

We will now discuss the garbage collection of metadata on both the client and server. This will enable a truly scalable implementation of CMVM-LV, where object metadata is kept at a minimum and deleted objects are actually deleted.

6.2.1 *Metadata Pruning*

The node history is the only data structure that cannot be pruned in any form, since it accurately describes which object versions were seen by the node. It can however be represented in a compact form, by a version vector together with bitmaps, which are kept small, due to the gap-filling by anti-entropy (as in DottedDB [22]). The other data structures can be completely removed given the right conditions.

Table 6 briefly describes the data structures in our framework that can be safely removed or compacted. Notice that for those that can be removed, the common knowledge required for their removal is to know when a version is replicated in all replica nodes for that key. This happens when the corresponding dot is in the node history of all those

Table 6: A list of data structures, their purpose and when they can be removed.

Structure	**S**cope	**P**urpose	**W**hen to Remove
Node History	Per node	Represent local object version history, as an extrinsic set	Never, but can be continuously compressed into constant-sized VV, plus some bitmaps to encode recent writes
Dot-Key Map	Per node	Used in anti-entropy to know which key a missing dot refers to	An entry can be removed when the dot is in all replica nodes for that key
Dependencies	Per object / session	Represent the causal past of the object or session	The non-maximal elements per key are not needed; any dependency can be removed when its dot is known by all replica nodes for that key
Key History	Per object / Per key in session	Represents the write history of that key	The history is not longer necessary when all current version dots (maximals) are known by all replica nodes for that key
Object	Per key	Contains the current versions, their dependencies and history	When the object only has null values, has no dependencies and no history, it can be deleted from storage

replica nodes. We will now describe how to learn this information, in order to remove unnecessary metadata.

6.2.1.1 *Stable Logical Time*

Node histories describe what the current local knowledge is. If each node caches information in the the node history received during anti-entropy from their peers in what concerns dots generated by itself, they can compute their *Stable Logical Time* (SLT): the highest counter c of dots from node i (i.e., dots (i, c)) that every peer has, such that they also have *every* other dot from i with a smaller counter[1].

Conceptually, the SLT_i of some node i is the highest counter c, such that $(i, c) \in \mathsf{NH}_i$ and the following property holds:

1 This is stronger than needed. A dot is fully replicated if it is present in all replica nodes of its key. For simplicity and performance, we chose rather to check if a dot in present in all peers.

$$\left(\bigcup_{k=1}^{c}(i,k)\right) \subseteq \left(\bigcap_{p \in \mathsf{peers}(i)} \mathsf{NH}_p\right)$$

To compute the stable logical time, a node i needs to know, for each peer p, the maximum counter c, such that p has in its history all dots from i up to c (i.e., dots $(i,1),\ldots,(i,c)$). To that end, the node state stores a new data structure SLT that maps peer ids to the respective maximum counter. This structure is defined for some node i as follows:

$$\mathsf{SLT}_i : \mathbb{I} \hookrightarrow \mathbb{N}$$

The stable logical time for some node i can now be computed as: $\min(\mathsf{ran}(\mathsf{SLT}_i))$.

6.2.1.2 *Causal Watermark*

In addition to maintaining the SLT data structure, each node gossips its stable logical time (i.e., the minimum counter) around the server, such that every node maintains a cache of stable logical times of all other nodes. We call this global view the *Causal Watermark* (WM), defined for some node i as:

$$\mathsf{WM}_i : \mathbb{I} \hookrightarrow \mathbb{N}$$

This data structure basically stores a lower-bound of the SLT for all nodes N in the server, such that:

$$\forall i, j \in N.\ \mathsf{WM}_i[j] \lessapprox \min(\mathsf{ran}(\mathsf{SLT}_j))$$

The staleness of the WM in relation to the real SLT depends on many factors such as the write throughput, the network conditions, the gossip interval and the anti-entropy interval.

6.2.2 *Client Library with GC*

The client library provides a Session object that encapsulates the context of a client session, enabling the client to execute multiple sessions in parallel without mixing dependencies.

The current definition of a session has a collection of dependencies (indexed by keys) and a map from keys to their history. As stated in Table 6, each dependency can be removed individually when it is present in all replica nodes, while non-maximals dependencies per key can also be removed. The history of a key can be removed when the maximals of that key are known by all replica nodes, but the maximals must be kept to be used in a subsequent write operation as the history of that write.

Therefore, a session now separates the maximals from the history, for the most recent read object for each key. A session object is redefined as follows:

$$\mathsf{Session} \doteq \mathbb{C} \times (\mathbb{K} \hookrightarrow (\mathbb{H} \times \mathbb{H}))$$

Algorithm 14 shows the updated operations in the client library. On the read operation, the only difference is the separation of the maximal dots and history of the object. Notice that the prior history information is overwritten by new reads, since the new history returned cannot be smaller than previous histories read (i.e., the client cannot read an older object after a newer object was returned in the same session).

The write operation first has to join the maximals with the history of the key, since the write will produce a new maximal element. The server returns a new dot that is added to the dependencies and placed in the key history as the new maximal. The key history is updated with the prior maximals and dependencies that are not maximals for that key are also removed, since it is sufficient to depend on the maximals elements to ensure that those versions are "visible" in a read operation.

Additionally, the client library has now an internal procedure to garbage collect a client session (see Algorithm 15). It first asks some random

Algorithm 14: Client Library with Garbage Collection support.

```
1  procedure Read((C, H) : Session, k : 𝕂):
2  |  (V, C′, H′) := rpc(k, Get, ⟨k, C[k]⟩)
3  |  C := C ⊔ C′
4  |  H[k] := (dom(V), H′)
5  |  return ((C, H), {v ∈ ran(V) | v ≠ null})
6
7  procedure Write((C, H) : Session, k : 𝕂, v : 𝕍):
8  |  H^k := fst(H[k]) ∪ snd(H[k])
9  |  d := rpc(k, Put, ⟨k, v, C, H^k⟩)
10 |  C[k] := {d} ∪ C[k] \ H^k
11 |  H[k] := ({d}, H^k)
12 |  return (C, H)
```

Algorithm 15: Garbage Collection operation in the Client Library.

```
1  function GC((C, H) : Session):
2  |  WM := rpc(random, GetWatermark, ⟨⟩)
3  |  for (k, C^k) ∈ C do
4  |  |  C[k] := { (j, c) ∈ C^k | c > WM[j] }
5  |  for (k, (H^k, _)) ∈ H do
6  |  |  if { (j, c) ∈ H^k | c > WM[j] } = ∅ then
7  |  |  |  H[k] := (H^k, ∅)
8  |  return (C, H)
```

node for its causal watermark. It then removes dependencies from the session that are now obsolete because they are in every node that replicates them. And finally, removes the history of any key whose current maximals are known by all replica nodes.

6.2.3 *Server Algorithms with GC*

In addition to the previous node state definition, each node now has: (a) a stable logical time SLT to represent the maximum contiguous knowledge of local dots in every peer; (b) a causal watermark WM to track a lower-bound, for each node, of its stable logical time; (c) a set NSK of *non-stripped keys* for local objects that still have non-stable causal history.

A server node state is initialized as follows:

Algorithm 16: Read request with GC support at Node i.

```
1 procedure Get(k : 𝕂, C_k : 𝒫(𝔻)):
2     (V, C, H) := fetch(k)
3     while (C_k ∪ C[k]) ⊈ NH_i do
4         o := rpc(k, fetch, ⟨k⟩)
          // the 'update' function updates the node history
5         (V, C, H) := update(k, o)
6     return (V, C, H)
```

Algorithm 17: Updated Store procedure at Node i.

```
1 procedure store(k : 𝕂, (V, C, H) : Object):
2     NH_i := NH_i ∪ H
3     for d ∈ dom(V) do
4         NH_i := NH_i ∪ {d}
5         DKM_i[d] := k
6     ST_i[k] := (V, C, H)
7     NSK_i := NSK_i ∪ {k}
```

$$\begin{aligned}\mathsf{Init}(i) = (&\mathsf{NH}_i := \emptyset, \mathsf{DKM}_i := \emptyset, \mathsf{ST}_i := \emptyset,\\ &\mathsf{SLT}_i := \emptyset, \mathsf{WM}_i := \emptyset, \mathsf{NSK}_i := \emptyset)\end{aligned}$$

6.2.3.1 *Reads*

Algorithm 16 shows the modified Get procedure with support for object garbage collection. The only change is in the loop condition, where the object history was replaced by the node history. The reason for this is that the object history is now removed when the current versions are in all replica nodes. Therefore, to test if the object and client dependencies are met by the local object, we must use the node history, which contains the exact history of the object, as stated by Invariant 1.

6.2.3.2 *Writes*

Regarding writing new versions, the only change from previous operations is in the store procedure in Algorithm 17, which now has a new last instruction to add the key to the NSK for later garbage collection.

Algorithm 18: Metadata GC operations at Node i.

```
1  process StripCausality():
2      loop forever
3          for k ∈ NSK_i do
4              objGC(k)
5          sleep(δ_s)
6
7  procedure objGC(k : 𝕂):
8      o := fetch(k)
9      (V, C, H) := o
10     for (k′, C^k) ∈ C do
11         C[k′] := { (j, c) ∈ C^k | c > WM[j] }
12     if { (j, c) ∈ dom(V) | c > WM[j] } = ∅ then
13         H := ∅
14     if C ∪ H = ∅ then
15         NSK_i := {k} ⩤ NSK_i
16     if (ran(V) \ {null}) ∪ C ∪ H = ∅ then
17         ST_i := {k} ⩤ ST_i
18     else if (V, C, H) ≠ o then
19         ST_i[k] := (V, C, H)
```

6.2.3.3 *Object Garbage Collection*

Algorithm 18 defines the object GC procedure objGC and the background process StripCausality, responsible for periodically iterating all keys in the NSK, calling objGC for each one.

The objGC procedure fetches the object from local storage and removes all dependencies of the object that are present in all replica nodes. It also removes the history if the current version dots are in all replica nodes. If the object does not have dependencies or history anymore, the key is removed from the NSK. Additionally, if there are only null values in the current versions and the dependencies and history are already removed, the object can be safely removed from storage. Otherwise, the local object is updated if anything changed.

6.2.3.4 *Anti-Entropy*

Algorithm 19 shows the modified procedures for the anti-entropy protocol. All the previous functionality is still present, but `SyncHistory` now

Algorithm 19: Anti-Entropy Protocol with GC support at Node i.

```
1  function SyncHistory(p : 𝕀, H : ℍ):
2      K := { k | (d, k) ∈ DKM_i ∧ d ∉ H ∧ p ∈ replicas(k) }
3      O := { (k, fetch(k)) | k ∈ K }
4      H^i := { (j, c) ∈ NH_i | j = i }
       // compute the maximum contiguous counter c from a p's dot in i's history
5      c := max({ c | c' ⩽ c ⇒ (p, c') ∈ NH_i })
6      async_rpc(p, SyncRepair, ⟨i, H^i, c, O⟩)
7
8  procedure SyncRepair(p : 𝕀, H^p : ℍ, c : ℕ, O : 𝕂 ↪ Object):
9      for (k, o) ∈ O do
10         update(k, o)
11     NH_i := NH_i ∪ H^p
       // update the Stable Logical Time and the Causal Watermark
12     SLT_i[p] := c
13     WM_i[i] := min(ran(SLT_i))
```

computes the highest counter corresponding to a dot from the asking node, in which the local node has that dot and all smaller dots in its history. This counter is sent to the node that initiated the anti-entropy protocol and is used in `SyncRepair` to first update the SLT map and then to recompute the causal watermark for local node entry.

6.2.3.5 *Watermark Gossip*

Algorithm 20 defines two operations for gossiping the watermark in the server. The GossipWatermark definition uses the simplest form of gossiping: randomly choose some node and send it the causal watermark. The receiving node updates the watermark by a pointwise maximum with the received watermark. Additionally, dots that are present in all replica nodes can be removed from the DKM, since they are not missing in any node and thus will not be used in future anti-entropy rounds.

6.3 ALTERNATIVE DESIGNS

In this section, we discuss some alternative design decisions, along with their advantages and disadvantages.

Algorithm 20: Watermark Gossip Operations at Node i.

```
1  process GossipWatermark():
2      loop forever
3          async_rpc(random, ReceiveWatermark, ⟨WM_i⟩)
4          sleep(δ_g)
5
6  procedure ReceiveWatermark(W : 𝕀 ↪ ℕ):
7      for j ∈ dom(W) do
8          WM_i[j] := max(WM_i[j], W[j])
       // remove DKM entries known by all peers
9      for (j, c) ∈ dom(DKM_i) do
10         if WM_i[j] ⩾ c then
11             DKM_i := {(j, c)} ⩤ DKM_i
```

6.3.1 *Derived Object Histories*

An alternative implementation would not store the object history in the object itself and instead rely on the node history. The object would only contain the current versions (maximals) and the dependencies (causal past), as follows:

$$\mathsf{Object} := (\mathbb{D} \hookrightarrow \mathbb{V}) \times \mathbb{C}$$

As stated in the Invariant 1, the node history is *extrinsic* to the union of all local objects history and current maximals. But the node history can only be represented in a compact way if it does not segment the histories per key, enabling compact representations like in DottedDB [22]. By not segmenting the node history per key, it loses the ability to find out what is the history of a particular key amongst the node history. For local operations, this is not a problem, since the additional histories from other keys do not interfere in the per-key versioning process, as object histories for different keys do not share dots.

Sending an object, either for replication or for a client read, becomes a problem, because the object history is absent from the object itself, being only diluted in the node history, but it should be attached to the object to be sent. Sending the node history attached to an object that is going to be replicated in another node (directly to that node or indirectly via

clients) is not a solution, because the receiving node cannot add the entire node history attached to the object to its own node history, since it would add versions of other objects that were not sent, violating the Node History Invariant 1.

What is needed is a way to *derive* the exact object history. If we take any object and only consider the dependencies on its key, the history of that object must be included in it, but not the other way around. However, if we intersect those self-dependencies with the node history, the result must be the exact history for that object, because the dependencies that were not part of the object history cannot be part of the node history, by definition. Therefore, we can use the object dependencies of its own key, to isolate dots in the node history that are from that object.

The *derived history* H^d of an object $o = (V, C)$, associated with the key k, and stored at node i, is defined as:

$$H^d(k, (V, C)) = NH_i \cap C[k]$$

We can now attach the derived history to an object sent to another node or to the client.

6.3.1.1 *Garbage Collection with Derived Histories*

Since an object no longer stores its history, the only object garbage collection needed is for its dependencies. In our previous implementation with explicit object histories, dependencies had two mechanisms to be removed: (1) only store the maximals of each key in the dependencies; (2) remove any individual dependency when they are present in every replica node for that key. The history was removed entirely when the maximals were known by all replica nodes.

But now, we use part of the dependencies in an object to derive its history. This means that removing those dependencies is semantically the same as removing the history. Therefore, for the object dependencies of its key, they must adopt the more pessimistic garbage collection rule of either dependencies and history, which is the latter.

For some object (V, C) associated with the key k, $C[k]$ is only removed when all current maximal dots $\mathsf{dom}(V)$ are in all replica nodes. The

other dependencies $C[k'], k' \neq k$, have the same GC mechanism as before.

ADVANTAGES The main advantage for using derived object histories is that each object never stores its history, relying on the (compact) node history. In our implementation, an object history is temporarily stored until it is garbage collected.

DISADVANTAGES One disadvantage of this approach is that the history of the object must be computed every time the object must be sent elsewhere. Also, the dependencies on its key cannot be garbage collected as fast as other dependencies, similar to the object history in the current implementation. This can actually be worse in some cases for space usage, because the history can be represented compactly as a version vector linear with the number of replica nodes, while dependencies must be individually represented, being unbounded in their number.

6.3.2 *Key-less Dependencies*

Although dependencies were defined as dots with a matching key, the latter is not mandatory: the dot uniquely identifies the version of an object that is a dependency for some client or object. Having the corresponding key allows, both the client library and the node serving a read request, knowing exactly what are the dependencies that correspond to versions of the key being read.

This can significantly reduces the amount of dependencies sent to the server when reading a key, or reduces the retrieved dependencies when a node serves a read request. The downside is the space that storing multiple keys may take, even-though we expect that dependencies will be garbage collected, using frequent anti-entropy and watermark gossip rounds.

Not storing the corresponding keys with dependencies allow for smaller metadata, but we lose the ability to precisely select dependencies of a particular key. As an alternative, we can filter dots in dependencies

whose node ids are from replica nodes of the target key. The result is obviously larger than the previous key filtering, but none-the-less can substantially reduce the amount of dependencies sent in a read request, or fetched by a node to serve a read request.

The alternative *key-less* dependencies are defined as follows:

$$\mathbb{C} \doteq \mathcal{P}(\mathbb{D})$$

In general, the modifications necessary to use key-less dependencies are: (a) implement dependencies as a set of dots instead of a map from keys to dots; (b) every time the dependencies where filtered for some key using $C[k]$, replace it with $\{(j, c) \in C \mid j \in \mathsf{replicas}(k)\}$.

6.4 FAULT-TOLERANCE

Given that our implementation uses a master-less design, there is no node that is a single point of failure. The replication factor R can be as large as desired, supporting $R - 1$ faults for every *replication group*: the set of nodes that replicate a common subset of keys. Clients can also fail without impacting the server or other clients.

Write operations always succeed if at least one replica node of the key is available to write. For a read operation on some key k, for every past dependency on that key, there must be at least one reachable node that has (or had) that dependency. Frequent anti-entropy rounds reduce the chance of some version not being available because some node failed or the network is partitioned.

Although not implemented, the server could take even further action to mitigate this problem, by tracking how many replicas of each version are there in the server. Thus, a version is only made visible to the client if it is *K-replicated*, where K is the number of replica nodes that have acknowledge to have that version. The downside of this approach is that it increases the latency for client write/delete operations.

6.5 DISCUSSION

We presented a distributed key-value store implementing Causal Multi-Value Consistency with Local-Versioning, allowing partial-replication with partial-overlapping, aiming for write-availabilty.

Most causally consistent data stores are designed to block on writes until the new object can be safely applied (i.e., the system is still causally consistent after the write). Our implementation took a different and less pessimistic approach, where writes never block and instead we rely on read operations to enforce causal consistency. This enables smaller client operation latency overall, since when a subsequent read is made to some node, that object could already be causally consistent. In exchange, some metadata must be temporarily attached to objects stored in the data store.

Our implementation frequently exchanges metadata, in order to garbage collect dependencies and other metadata as fast as possible. The implementation is based on our previous work Node-wide Dot-based Clocks. We expect that a real implementation of our new consistency model to perform well, given the results of the evaluation of DottedDB.

7

TRANSACTIONAL CAUSAL MULTI-VALUE CONSISTENCY

Although causal consistency provides client-friendly session guarantees, there are cases where its not enough to execute a sequence of operations in a causal session and obtain a correct outcome. Specifically, multiple read operations executed in sequence have no guarantees about what can happen between them, opening the door for having an inconsistent state as a whole, even though the individual reads respect causality. Multiple writes have the same problem, where a client wants to write two items that only make sense when presented together. Writing the items in sequence in a causally consistent manner is not enough, because after the first write returns (but before the second is performed), another client can read the first write and then read an older version of the second item.

To solve these issues, we now extend the Causal Multi-Value Memory to support two additional operations: read-only transactions and write-only transactions. We first provide specific examples where read-only and write-only transactions are needed, then we formally define the extended consistency model and finally provide an implementation, which extends our previous implementation.

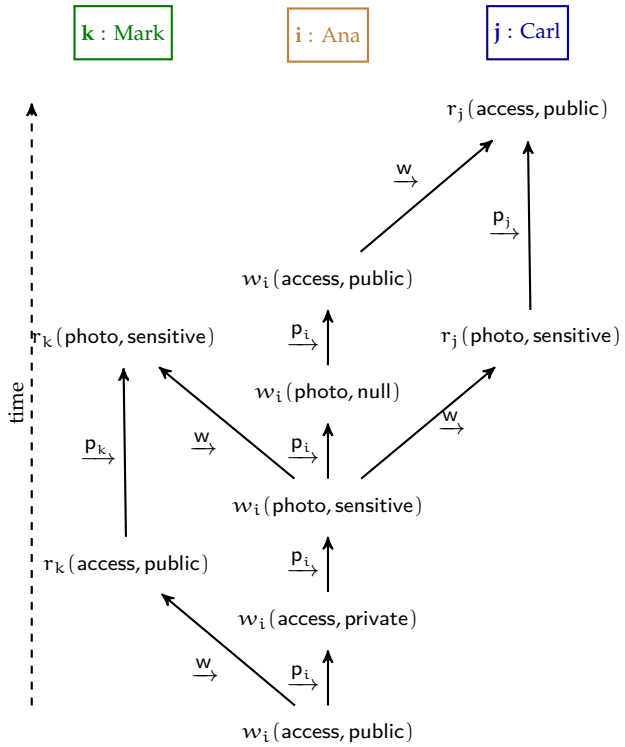


Figure 23: Ana makes several updates to a particular photo and its access level. Two friends, Mark and Carl, try to see the photo and check the access level in different orders, and both see a sensitive photo from Ana with public access.

7.1 MOTIVATION

7.1.1 *Read-only Transactions*

Consider a scenario where Ana is using a social network and has an album of photos and associated with that album an access control setting. Figure 23 shows a scenario where Ana does several operations: setting the album private, uploading a sensitive photo, deleting the sensitive photo and setting the access to public again. Also represented are two of Ana's friends that logged in and want to see updates to Ana's album. The two friends do the same read operations (read the access control setting and read a photo), but in different order. As we can see, there is no order in which the social network application could access Ana's operation history that guarantees not returning the sensitive photo with a public access.

This particular example could be solved with read transactions for both Carl and Mark. In Carl's case, reading the latest public access control, ensured that it had a dependency on the latest photo write, which makes the transaction return a `null` photo. In Mark's case, reading the sensitive photo ensures that it has a dependency on the private access write, making the read transaction return that access control and not a previous one. Figure 24 shows the same scenario with read transactions and the two different, but consistent, outcomes.

7.1.2 *Write-only Transactions*

Write-only transactions have many applications. Continuing in our social network example, if a user updates a photo and updates its description, we want both updates to depend on each other. Without a write transaction, a write order must be chosen and depending on that, another user can see one update without necessarily seeing the other. Another example is friendship requests, where a write transaction guarantees that if A is seen as a friend of B, then B must be seen as a friend of A, and vice-versa.

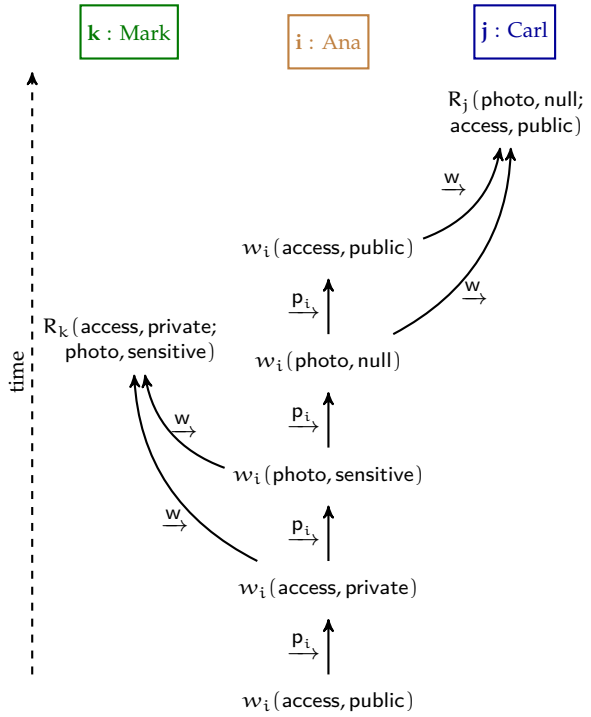


Figure 24: Ana makes several updates to a particular photo and its access level. Both Mark and Carl perform a read transaction and obtain different results, albeit both acceptable and consistent with Ana's operations.

7.2 TRANSACTIONAL CAUSAL MULTI-VALUE MEMORY

This model is an extension, or a generalization, of the previous Causal Multi-Value Memory. There are now four types of operations available:

- $w_i^m(x, v)$: a write to location x of value v, being the m^{th} operation by p_i;
- $r_i^m(x, V)$: a read from location x, returning a set of values V, being the m^{th} operation by p_i;
- $R_i^m(S_r)$: a read transaction containing a map S_r associating each key read to the set of values returned, being the m^{th} operation by p_i;
- $W_i^m(S_w)$: a write transaction containing a map S_w associating each key to the written value, being the m^{th} operation by p_i;

A read operation can be seen as a specific case of a read transaction with a singe read operation: $r(x, V) \Leftrightarrow R(\{x \mapsto V\})$. A write operation can also be seen as a specific case of a write transaction containing only that write: $w(x, v) \Leftrightarrow W(\{x \mapsto v\})$. Thus, hereafter we will focus on the more general case, involving the transactional operations.

7.2.1 *Ordering*

7.2.1.1 *Process Order*

Process Order $\xrightarrow{p_i}$ is a binary relation on operations of the execution history H_i of process p_i, according to p_i local time:

$$o_i^m \xrightarrow{p_i} o_i^n \doteq m < n$$

The global process order relation $\xrightarrow{p}$ is the union of the process orders of all processes:

$$\xrightarrow{p} \doteq \bigcup_{i \in \mathbb{I}} \xrightarrow{p_i}$$

7.2.1.2 *Writes-Into Order*

Writes-into order is a binary relation induced by history H, relating writes and reads on the same location, defined by:

$$W(S_w) \xrightarrow{w} R(S_r) \doteq \exists (x, v) \in S_w . v \in S_r[x]$$

7.2.1.3 *Causality Order*

Causality Order ($\xrightarrow{c}$) is the relation formed by the transitive closure of the union of process order ($\xrightarrow{p}$) and writes-into order ($\xrightarrow{w}$):

$$\xrightarrow{c} \doteq \left(\xrightarrow{w} \cup \xrightarrow{p} \right)^{+}$$

7.2.1.4 *Causal Past*

The *causal past* of an operation o for some location x can be defined by the function $C^x(o)$, which returns the set of operations that explain all values written to x before o, according to the causality order:

$$C^x(o) \doteq \{ W(S_w) \mid W(S_w) \xrightarrow{c} o \wedge x \in \mathsf{dom}(S_w) \}$$

The entire causal past of an operation o is defined by the union of the causal past of o for all locations:

$$C(o) \doteq \bigcup_{x \in L} C^x(o)$$

7.2.2 *Transactional Causal Multi-Value Histories*

A history H is *Transactional Causal Multi-Value* (TxCMV) if and only if it respects the following three properties:

- *No cycles*: $\xrightarrow{c}$ induced by H is a strict partial order;
- *Read Some Write*:

$$R(S_r) \in H \Rightarrow \forall (x, V) \in S_r . \forall v \in V . \exists W(S_w) \in H . (x, v) \in S_w$$

- *Read Last Writes*: presented bellow in two variants.

The first condition forbids $\xrightarrow{c}$ cycles, preventing "reads from the future", or *out of thin air* values, which explain themselves in a causal loop. The second condition ensures that each value in the set returned by a read comes from some write to the corresponding location. The third condition dictates which values a read operation must return. There are two variants of this condition: a *global versioning* and a *per-location versioning*.

7.2.2.1 *Transactional Causal Multi-Value with Global Versioning Histories*

A history is Transactional Causal Multi-Value with *Global Versioning* (TxCMV-GV) if in addition to the first two conditions, it respects the following:

- *Read Last Writes with Global Versioning*:

$$o = R(S_r) \Rightarrow \forall (x, V) \in S_r.\, V = \{\, S_w[x] \mid W(S_w) \in \max(C^x(o), \xrightarrow{c}) \,\} \tag{7}$$

It states that for a given operation o that reads from some locations, for each such location x, given the set of operations that write to x that are in the past of o according to the causality order, then o must return all values written to x by the maximal elements of that set.

TRANSACTIONAL CAUSAL MULTI-VALUE WITH GLOBAL VERSIONING MEMORY A memory is TxCMV-GV if it only admits TxCMV-GV histories.

To define Transactional Causal Multi-Value with per-Location Versioning (TxCMV-LV) Histories, we have to introduce a new per-location writes-into order and the resulting per-location history order.

7.2.3 *Transactional Causal Multi-Value with per-Location Versioning Histories*

7.2.3.1 *Per-Location Writes-Into Order*

Writes-into order for a specific location x is a binary relation induced by history H, relating writes and reads on x, defined by:

$$W(S_w) \xrightarrow{w^x} R(S_r) \doteq \exists v.\, (x, v) \in S_w \wedge v \in S_r[x]$$

The union of writes-into order for all locations in L is equal to the global writes-into order defined previously:

$$\bigcup_{x \in L} \xrightarrow{w^x} = \xrightarrow{w}$$

7.2.3.2 *Per-Location History Order*

History Order for some location x ($\xrightarrow{h^x}$) is the relation formed by the transitive closure of the union of process order ($\xrightarrow{p}$) and writes-into order for x ($\xrightarrow{w^x}$):

$$\xrightarrow{h^x} \doteq \left(\xrightarrow{w^x} \cup \xrightarrow{p} \right)^+$$

Since $\xrightarrow{w^x}$ is a subset of $\xrightarrow{w}$, each location's history order is also a subset of the causality order:

$$\xrightarrow{h^x} \subseteq \xrightarrow{c}$$

7.2.3.3 *Transactional Causal Multi-Value with per-Location Versioning Histories*

A history is Transactional Causal Multi-Value with *per-Location Versioning* (TxCMV-LV) if in addition to the previous two *No cycles* and *Read Some Write*, it respects the following:

- *Read Last Writes with per-Location Versioning*:

$$o = R(S_r) \Rightarrow \forall (x, V) \in S_r.\, V = \{\, S_w[x] \mid W(S_w) \in \max(C^x(o), \xrightarrow{h^x}) \,\} \tag{8}$$

The difference between global versioning (Condition 7) and per-location versioning (Condition 8) is that the former takes the maximal elements of the causal past of a read according to the *global causality order*, while the latter takes the maximals of the same causal past, but according to the location's *history order*.

TRANSACTIONAL CAUSAL MULTI-VALUE WITH PER-LOCATION VERSIONING MEMORY A memory is TxCMV-LV if it only admits TxCMV-LV histories.

7.2.4 *Transactional CMV versus Non-Transactional CMV Memories*

It can be seen that if all operations in a TxCMV memory are to a single location, its definition corresponds exactly to what was previously defined in CMV memories.

7.3 DISTRIBUTED TRANSACTIONAL CAUSAL MULTI-VALUE DATA STORE

We now define an implementation of Transactional CMV-LV memory in a distributed key-value store. This implementation is an extension of the previous implementation of CMV-LV memory, preserving all its operations unless redefined hereafter.

7.3.1 *Client Library*

Algorithm 21 shows the two new operations available in the client library.

Algorithm 21: Client Library for Transactional Operations.

```
1  procedure ReadTx((C, H) : Session, K : 𝒫(𝕂)):
2      C_K := {(k, D) ∈ C | k ∈ K}
3      O := rpc(random, GetTransaction, ⟨K, C_K⟩)
4      for (k, (V, C′, H′)) ∈ O do
5          C := C ⊔ C′
6          H[k] := (dom(V), H′)
7          O[k] := {v ∈ ran(V) | v ≠ null}
8      return ((C, H), O)
9
10 procedure WriteTx((C, H) : Session, W : 𝕂 ↪ 𝕍):
11     for (k, v) ∈ W do
12         W[k] := (v, fst(H[k]) ∪ snd(H[k]))
13     (t, D) := rpc(random, PutTxPrepare, ⟨W, C⟩)
14     while ok ≠ AwaitTxCommit(t) do
15         rpc(random, PutTxCommit, ⟨t, dom(W), C ⊔ D⟩)
16     for (k, {d}) ∈ D do
17         H^k := snd(W[k])
18         C[k] := {d} ∪ C[k] \ H^k
19         H[k] := ({d}, H^k)
20     return (C, H)
```

READ TRANSACTIONS The read transaction (ReadTx) takes a set of keys and returns a set from objects consistent with TxCMV-LV, i.e., the server cannot return objects that have future versions either in the session dependencies or in the dependencies of any object returned. The client library only sends to the server the dependencies that correspond to the keys being read. The server returns a map of keys to objects. All object dependencies and histories are added to the session metadata. The current versions dots are separated from the object history, to enable future garbage collection.

WRITE TRANSACTIONS The write transaction (WriteTx) takes a map from keys to values and writes to the server. A key can also map to a special null value to represent a delete. All new object versions created in the server will have dependencies on each other. All session dependencies are sent to the server, as well as histories of the keys being written.

The client library first calls PutTxPrepare at some server node to act as the coordinator that asks for new dots for every write in the transaction. When the coordinator has a new dot for each write, it returns the unique transaction ID and the new dependencies. This intermediate result is for fault-tolerance only, because it enables the client library to retry the *commit* phase of the transaction in the server, without repeating the *prepare* phase and creating new dots.

The client waits for the confirmation that all writes were "committed" using the function AwaitTxCommit(t), that returns ok when everything is done for transaction t or it gives a timeout, which makes the client retry the commit phase to another server node, using the PutTxCommit operation. When the confirmation is successful, the new dependencies are added to the session and the histories are updated similarly to single-key write operations.

7.3.2 *Server Algorithms*

7.3.2.1 *Read-only Transaction*

In a read transaction, we ideally want to retrieve the exact object versions corresponding to the missing dependencies/dots, to avoid reading newer versions which can themselves have other newer dependencies. To that end, we introduce a new data structure that temporarily stores objects corresponding to dots that were obsoleted. This gives an opportunity to running read transactions of fetching the exact version of an object. This structure is called *Temporary Object Cache* (TOC), mapping dots to a tuple containing the object and the timestamp of when they were "removed". It is defined as follows:

$$\mathsf{TOC} : \mathbb{D} \hookrightarrow (\mathsf{Object} \times \mathbb{N})$$

Algorithm 22 defines the read transaction and the auxiliary function fetch_dot. A read transaction maintains two main variables: H - represents both the history and maximals of current read objects; C - represents the client dependencies plus the current objects' dependencies. While the dependencies in C are not met in H, the transaction keeps

Algorithm 22: Read Transaction at Node i.

```
1  function GetTransaction(K : 𝒫(𝕂), C : ℂ):
2      O := ∅ ; H := ∅
3      while C ⋤ H do
4          parallel for (k, D) ∈ (C − H) do
5              (o, D′) := rpc(k, fetch_dot, ⟨k, D⟩)
6              (_, C′, H′) := o
7              H[k] := H[k] ∪ H′ ∪ D′
8              C := C ⊔ C′|_K
9              O[k] := merge(O[k], o)
10     return O
11
12 function fetch_dot(k : 𝕂, D : 𝒫(𝔻)):
13     o := (∅, ∅, ∅)
       // try to read old objects for the requested dots
14     for d ∈ D ∩ dom(TOC_i) do
15         (o′, _) := TOC_i[d]
16         o := merge(o, o′)
       // if there are no old objects, read the current one
17     if o = (∅, ∅, ∅) then
18         o := fetch(k)
       // return the object and the dots known by this node
19     return (o, D ∩ NH_i)
```

trying to fetch the *exact* versions for missing dependencies, using the function fetch_dot.

This function tries to return an object that reflects the exact dots asked using the TOC, to avoid introducing additional dependencies. If the asked dots are not in the TOC, it is because they are still the current versions, therefore the local object is returned. Additionally, the function returns the dots that the node knows about in its history, in case the object history was already garbage collected.

To support the new TOC data structure, the update function was modified to add any object whose current version was removed to the TOC. The new definition can be found in Algorithm 23.

GC FOR TEMPORARY OBJECT CACHE A new background process named TOC_GC was introduce to garbage collect the cache of old ob-

Algorithm 23: Update operation with TOC support at Node i.

```
1 procedure update(k : 𝕂, (V, C, H) : Object):
2     o := fetch(k)
3     if dom(V) ⊈ NH_i then
4         (V, C, H) := merge((V, C, H), o)
          // save objects whose dots where obsoleted
5         for (d, v) ∈ {ver ∈ fst(o) | ver ∉ V} do
6             TOC_i[d] := (o, now())
7         o := (V, C, H)
8         store(k, o)
9     return o
```

Algorithm 24: Garbage Collection for TOC at Node i.

```
1 process TOC_GC():
2     loop forever
3         TOC_i := {(d, (o, ts)) ∈ TOC_i | (now() − ts) < δ_r}
4         sleep(δ_r)
```

jects, when they are too old. Its definition is found in Algorithm 24. The δ_r value should match the maximum time a read transaction can run.

7.3.2.2 *Write-only Transaction*

Ideally, a replica node of one of the keys in the write set should be the transaction coordinator, but any node can coordinate. The coordinator receives a map from keys to values and histories to write, in addition to the client dependencies. This operation must ensure that each new object depends on all other versions created by this transaction.

The coordinator will execute a protocol similar to *two-phase commit* [23], except that cohorts (replica nodes of keys involved in the transaction) always reply positively with a new dot. The main idea is to divide the transaction in two phases: (1) *prepare*: create a transaction ID, send the new values to replica nodes of that key and receive a new dot for every write in the transaction; (2) *commit*: send the new dots to all replica nodes of keys in the transaction to commit the new objects, making them visible to read operations.

Before being committed, cohort nodes keep the new transaction state in a new data structure called *Quarantine Writes* (QW): a map from a transaction ID to another map from keys to their new values, new dots and their corresponding histories. This structure is defined as follows:

$$\mathsf{QW} : \mathbb{D} \hookrightarrow \mathbb{K} \hookrightarrow (\mathbb{D} \times \mathbb{V} \times \mathbb{H})$$

The transaction ID is defined as dot, which is generated using a new node *Transaction Counter* (TXC): an incremental counter starting at zero for every node.

PREPARE The PutTxPrepare procedure in Algorithm 25 executes the prepare phase of the write transaction. The coordinator first creates a new unique transaction ID using the TXC counter. Then, for each write it sends the transaction ID, the key, new value and the history to *one* replica node, using the prepare operation. This is done is parallel since each request can be made independently.

The prepare operation (defined in Algorithm 26) in a replica node first tests if the transaction ID and the key are already present in the quarantine writes. If this was the first time a prepared was issued by the coordinator for this transaction and key, the entry will not exist and the node will create a new dot for the information received. However, because clients can retry the prepare phase, the entry for this transaction and this key can already exist in the node. In that case, the node simple reuses the already generated dot.

The dot, along with the transaction information for that write, are sent to all replica nodes for that key to execute the prepared function, which first tests if it has already seen this transaction ID/key pair. If not, it just adds that information for a future commit message. If the node already has the that entry in the QW, then it must choose which entry to keep in a deterministic way, to avoid having inconsistent entries across replica nodes.

The coordinator upon receiving a new dot for each key in the transaction, sends the transaction ID and the new dependencies to the client to enable it to retry the commit phase later, without repeating the pre-

Algorithm 25: Write-only Transaction API at Node i.

```
1  procedure PutTxPrepare(W : 𝕂 ↪ (𝕍 × ℍ), C : ℂ):
2      TXC_i := TXC_i + 1
3      t := (i, TXC_i)
4      D := ∅
5      parallel for (k, (v, H)) ∈ W do
6          d := rpc(k, prepare, ⟨t, k, v, H⟩)
7          D[k] := {d}
8      send_client(t, D)
9      PutTxCommit(t, dom(W), C ⊔ D)
10
11 procedure PutTxCommit(t : 𝔻, K : 𝒫(𝕂), C : ℂ):
12     for j ∈ ⋃{ replicas(k) | k ∈ K } do
13         async_rpc(j, commit, ⟨t, C⟩)
14     return ok
```

pare phase. This is strictly for fault-tolerance. Finally, the coordinator executes the second phase, calling the PutTxCommit procedure with the transaction ID, the keys involved in the transaction and the updated dependencies.

COMMIT The PutTxCommit procedure in Algorithm 25 executes the second phase of the write transaction. The coordinator asynchronously calls the commit operation with the transaction ID and the updated dependencies, for every node who is a replica node of at least one key in the transaction.

The commit procedure in Algorithm 26 on a replica node takes every entry in the QW for that transaction ID and updates the local storage with the new information. Each updated object has a dependency for all other new versions created by that transaction. After all entries are updated and merged to local "non-quarantine" objects, the transaction ID entry can be removed from the QW and it can return a successful acknowledgement. If the node did not have quarantine writes for the received transaction ID, it returns an unsuccessful acknowledgement to the coordinator.

Algorithm 26: Auxiliary Operations for Write-only Transactions at Node i.

```
1  procedure prepare(t : 𝔻, k : 𝕂, v : 𝕍, H : ℍ):
2      if t ∉ dom(QW_i) ∨ k ∉ dom(QW_i[t]) then
3          d := (i, max({ c | (n, c) ∈ NH_i ∧ n = i }) + 1)
4          NH_i := NH_i ∪ {d}
5      else
           // reuse the dot genererated from previous attempts of prepare
6          (d, _, _) := QW_i[t][k]
7      for j ∈ replicas(k) do
8          async_rpc(j, prepared, ⟨t, k, d, v, H⟩)
9      return d
10
11 procedure prepared(t : 𝔻, k : 𝕂, d : 𝔻, v : 𝕍, H : ℍ):
12     if t ∉ dom(QW_i) ∨ k ∉ dom(QW_i[t]) then
13         QW_i[t][k] := (d, v, H)
14     else
           // if there is already an entry for this transaction, deterministically choose
              the same winning dot using a total order on the node IDs
15         (d', _, _) := QW_i[t][k]
16         if fst(d) > fst(d') then
17             QW_i[t][k] := (d, v, H)
18
19 procedure commit(t : 𝔻, C : ℂ):
20     if t ∉ dom(QW_i) then return error
21     for (k, (d, v, H)) ∈ QW_i[t] do
22         update(k, ({(d, v)}, C, H))
23     QW_i := {t} ⩤ QW_i
24     return ok
```

ANTI-ENTROPY WITH QUARANTINE WRITES New dots are created and added to the node history for quarantine writes that are not yet normal objects ready to replicate. Therefore, the anti-entropy protocol must be careful in sending the node history to another node, because in the current implementation, the receiving node i assumes that all dots from j sent by j in a anti-entropy round without a corresponding object also sent, are from keys not replicated by i and adds those dots to i's node history explicitly, closing any gaps in node history for j's dots which enables a compact node history representation.

Algorithm 27: Anti-Entropy Operation with Quarantine Writes at Node i.

1 **function** `SyncHistory`$(p : \mathbb{I}, H : \mathbb{H})$**:**
2 $\quad K := \{ k \mid (k, d) \in \mathsf{DKM}_i \wedge d \notin H \wedge p \in \mathsf{replicas}(k) \}$
3 $\quad O := \{ (k, \mathsf{fetch}(k)) \mid k \in K \}$
4 $\quad H^i := \{ (j, c) \in \mathsf{NH}_i \mid j = i \}$
$\quad$ `// remove dots from quarantine versions from the set` H^i
5 $\quad H^i := H^i \setminus \bigcup \{ \{ d \mid (k, (d, v, H)) \in W \} \mid (t, W) \in \mathsf{QW}_i \}$
6 $\quad c := \max(\{ c \mid c' \leqslant c \Rightarrow (p, c') \in \mathsf{NH}_i \})$
7 $\quad \mathsf{async\_rpc}(p, \texttt{SyncRepair}, \langle i, H^i, c, O \rangle)$

This is incorrect for quarantine writes, since some of those dots without associated objects sent from j may be from the quarantine writes in j that are not yet objects and thus are not present in the DKM. If i added the dots from quarantine writes in j to its node history, those dots would not be requested in future AE rounds.

The solution is to filter dots that correspond to quarantine writes in QW from the node history sent to i, as defined in Line 4 of `SyncHistory` procedure in Algorithm 27. This way, those dots will not be added to i's node history and subsequent anti-entropy rounds will ask again for those dots if they do not arrive through normal commit propagation, eventually replicating the updated objects from committed quarantine writes.

GC FOR QUARANTINE WRITES Entries are removed from QW when their transaction ID is committed. However, messages can be lost or network partitions can occur. To avoid having lingering writes in a quarantine state when they were already committed in other replica nodes, the StripCausality process can check if any recently updated object (in the NSK) has a dot from a quarantine write in either its history or in one of the current versions. If either one is true, then the quarantine write was committed by another replica node and that entry is no longer necessary.

Algorithm 28 defines the updated StripCausality process, which collects the dots from the non-stripped keys and filters quarantine writes that are in that set.

Algorithm 28: Metadata GC operations with Quarantine Writes GC support at Node i.

```
1  process StripCausality():
2      loop forever
3          D = ∅
4          for k ∈ NSK_i do
5              (V, C, H) := fetch(k)
6              D := D ∪ dom(V) ∪ H
7              objGC(k)
8          for (t, W) ∈ QW_i do
9              QW_i[t] := { (k, (d, v, H)) ∈ W | d ∉ D }
10         sleep(δ_s)
```

There is still a chance that some entries in the QW will not be garbage collected in the described algorithm, because entries with dots that were never committed and were never replaced by the newer dots (created in retries of the prepare phase by the client), won't receive objects corresponding to those old dots. In a similar approach to the TOC garbage collection used in read transactions, the node can ultimately apply a *TTL* policy to entries that are much older than what a write transaction can run without aborting due to a timeout.

7.4 DISCUSSION

We extended our previous Causal Multi-Value Memory model to provide read-only and write-only transactions.

Since read-only transactions run at the same time as other client operations in other nodes, data can be changing during the execution of the transaction. To avoid unbounded rounds of data fetches due to new dependencies being introduced, the data store needs to keep older versions of a key temporarily available. This enables the transaction to return slightly older data that is causally consistent with other data being returned in the transaction. The garbage collection of older data is time-based and automatic, which means that it does not introduce a significant overhead on nodes.

Write-only transactions involve a two step protocol in their execution. Each individual write in the transaction needs a unique id (dot) and therefore the first phase is responsible for creating and grouping all of the new dots. The second phase commits each individual write by adding all new dots as its dependencies. This way, every object created in the transaction has a dependency of every other write in the transaction. This is useful for cases where the client wants to write to multiple keys that mutually depend on each other, and need *atomic visibility* of updates to them, something not achievable having only read-transactions.

7.4.1 *Fault-Tolerance*

7.4.1.1 *Read-only Transactions*

In a read transaction, at least one replica node of every key should be available to fetch an object. If a second round is necessary, there must be at least one replica node that has the unique version being requested. Also, the total read transaction time should be smaller than the δ_r time used in the TOC garbage collection, to ensure that any older version is still available at replica nodes. If it exceeds this time, the transaction should be restarted.

7.4.1.2 *Write-only Transactions*

Regarding write-only transactions, the transaction ID generated by the coordinator enables the client to retry the transaction commit phase multiple times, avoiding the generation of concurrent versions for the same write in the transaction. Given that the prepare procedure checks if there is already a dot generated for the transaction ID/key pair, a concurrent version can only appear if: (a) the client restarts the same transaction without reusing the transaction ID returned before by the coordinator, or (b) the client reuses the same transaction ID, but the coordinator repeats the prepare invocation to a different replica node that has not seen the transaction ID/key in the QW.

Even in the last case, no matter what the order the prepared messages for different prepare attempts arrive to replica nodes, the QW converges to a deterministic entry for that transaction ID, since the prepared procedure always chooses the same winner in case of *conflict*, using the total order on the node ids in the different dots.

To increase the write transaction fault tolerance: (a) the nodes executing the prepare procedure could require a larger number of acknowledgements from the prepared operations sent to other replica nodes, before responding to the coordinator; (b) the coordinator could wait for at least one *successful* commit acknowledgement from a replica node of each key in the transaction.

8

CONCLUSION

We explored in this thesis the topic of causality in distributed key-value stores. To respect causality, these systems must provide a multi-value API to clients. Clients write single values to keys, but a read operation can return more that one value associated with a key, if those values were written concurrently according to causality.

Traditional logical clocks like Versions Vectors can be used to provide multi-value APIs in distributed data stores, but fail to capture per-write causal information necessary to correctly resolve concurrent writes. This lead to false concurrency being introduced to the system, in what was coined as "sibling explosion" [1] by some practitioners. We proposed *Dotted Version Vectors* that correctly captures per-write causality, avoiding this issue in multi-value data stores, without extra size or computational complexity.

Dotted Version Vectors correctly capture causally, but they still have to be stored forever in the data store to respect causality. This can be problematic if nodes are constantly being replaced, introducing newer identifiers that pollute metadata. We took the lessons learned in Dotted Version Vectors and presented *Node-wide Dot-based Clocks*, a novel framework that resumes information common to all keys in a node to a single logical clock per node. This allowed most metadata to be garbage collected, with the side-effect of allowing metadata-free distributed deletes in the data store. Another benefit of this framework is a lightweight anti-entropy protocol that replaces the Merkle Trees as the main mechanism for repairing out-of-sync data between replica nodes.

1 Sibling is a term that was given to causally concurrent versions of the same key.

Our implementation of Node-wide Dot-based Clocks in DottedDB was evaluated against a similar data store prototype, with the current state of the art mechanisms. The results show a clear improvement of the anti-entropy protocol and an overall reduced metadata for causality tracking.

Most distributed data stores still only offer weak consistency guarantees in order to achieve high-availability and performance. Eventual consistency is the weakest form of consistent available in most of these systems. However, causal consistency can be implemented in a highly-available system and offers many advantages from a clients perspective like sessions guarantees. Even-though causal consistent is said to be the strongest consistency model that still supports high-availability, it does not support a multi-value API. Most causally consistent systems either arbitrate a winner when a conflict occurs (e.g. choosing the write with the latest timestamp) or use special data-types like CRDTs that automatically merge conflicting writes. The former option leads to data loss and the latter imposes the use of specific data-types with their own trade-offs and overhead.

We defined a new consistency model named *Causal Multi-Value Consistency* that respects causality across all keys and still supports a multi-value API. We explored different variants and compared them to causal consistency. We also provided a reference implementation of this novel consistency model in a distributed key-value store, building on top of our Node-wide Dot-based Clocks framework. In addition, given that sometimes multi-key operations are needed to correctly express the client intent, we extended our new consistency model to include read-only and write-only transactions. A reference implementation was also provided.

8.1 FUTURE WORK

As future work, we believe there are several paths worth of further research. One of them is implementing a distributed key-value store that supports Causal Multi-Value Consistency. We proposed and defined

the main algorithms for such implementation, but a real implementation could be evaluated against similar causal systems, showing more clearly the trade-offs of our solution.

Another path of research is to explore general read/write transactions in the context of the Transactional Causal Multi-Value Memory model. We only defined *Read-only* and *Write-only* transactions, but general transactions should at least be partially supported. For example, if every read operation in a general transaction can be executed before write operations, the entire transaction could be achieved by composing a read-only with a write-only transaction.

A

MATHEMATICAL NOTATION

A.1 SETS

We use mostly standard notation for sets and maps, including set comprehension of the form $\{ f(x) \mid x \in S \}$ or $\{ x \in S \mid \mathtt{Pred}(x) \}$. We also use $\mathcal{P}(s)$ for the power set of some set s.

A.1.1 *Maximal Elements in a Partially Ordered Set*

An element of a partially ordered set is *maximal* under a given order, if it is not smaller than any other element, according to that order:

$$\max(S, <) \doteq \{ e \in S \mid \nexists e' \in S . e < e' \}$$

A.1.2 *Pre-defined Sets*

We use $\mathbb{N}$ for natural numbers, and also $\mathbb{I}$, $\mathbb{K}$ and $\mathbb{V}$ for some set of node identifiers, keys and values, respectively.

A.2 MAPS

A map is a set of (k, v) pairs, where each k is associated with a single v. Given a map m, $m[k]$ returns the value associated with key k, while $m[k] := v$ updates the value associated with k with v.

A.2.1 *Bottom Values*

For convenience we also use $m[k]$ when $k \notin \mathsf{dom}(m)$ and B has a bottom, to denote $\bot_B$. For example, for some map $m : \mathbb{I} \hookrightarrow \mathbb{N}$, then $m[k]$ denotes 0 for any unmapped key k.

A.2.2 *Domain and Range*

The domain and range of a map m is denoted by $\mathsf{dom}(m)$ and $\mathsf{ran}(m)$, respectively. They can be defined as follows:

$$\mathsf{dom}(m) = \{\, k \mid (k, v) \in m \,\}$$
$$\mathsf{ran}(m) = \{\, v \mid (k, v) \in m \,\}$$

A.2.3 *Domain Subtraction*

We use $\lhd$ for domain subtraction, where $A \lhd B$ is the map obtained by removing from B all pairs with a key that is included in the domain of A. It can be defined as follows:

$$A \lhd B = \{\, (k, v) \in B \mid k \notin \mathsf{dom}(A) \,\}$$

A.2.4 *Map Subtraction*

We denote as $A - B$ the subtraction between two maps, and can be defined as follows:

$$A - B = \{\, (k, V \setminus B[k]) \mid (k, V) \in A \wedge V \setminus B[k] \neq \emptyset \,\}$$

A.2.5 *Domain Restriction*

We use $M\big|_K$ to denote the domain restriction of map M to the set K. It is defined as:

$$M\big|_K = \{\, (k, v) \in M \mid k \in K \,\}$$

A.2.6 *Merging Maps*

We denote the merge of two maps A and B as $A \sqcup B$, where their common keys have their values merged.

$$\begin{aligned} A \sqcup B = &\{\, (k, A[k] \cup B[k]) \mid k \in \mathsf{dom}(A) \cap \mathsf{dom}(B) \,\} \\ &\cup \{\, (k, v) \in A \mid k \notin \mathsf{dom}(B) \,\} \\ &\cup \{\, (k, v) \in B \mid k \notin \mathsf{dom}(A) \,\} \end{aligned}$$

A.2.7 *Partial Map*

We use $A \hookrightarrow B$ for a partial function from A to B. Given such a map m, then $\mathsf{dom}(m) \subseteq A$ and $\mathsf{ran}(m) \subseteq B$.

A.3 PAIRS

We use $\mathsf{fst}(p)$ and $\mathsf{snd}(p)$ to denote the first and second component of a pair p, respectively.

BIBLIOGRAPHY

[1] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. "Causal memory: Definitions, implementation, and programming." In: *Distributed Computing* 9.1 (1995), pp. 37–49.

[2] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. "Bounded Version Vectors." In: *DISC*. Ed. by Rachid Guerraoui. Vol. 3274. Lecture Notes in Computer Science. Springer, 2004, pp. 102–116. ISBN: 3-540-23306-7.

[3] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. "Interval Tree Clocks." In: *Proceedings of the 12th International Conference on Principles of Distributed Systems*. OPODIS '08. Luxor, Egypt: Springer-Verlag, 2008, pp. 259–274. ISBN: 978-3-540-92220-9.

[4] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. "Scalable and accurate causality tracking for eventually consistent stores." In: *Distributed Applications and Interoperable Systems*. Springer. 2014, pp. 67–81.

[5] Sérgio Almeida, João Leitão, and Luís Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 85–98.

[6] Hagit Attiya, Faith Ellen, and Adam Morrison. "Limitations of Highly-Available Eventually-Consistent Data Stores." In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*. 2015, pp. 385–394. DOI: 10.1145/2767386.2767419. URL: http://doi.acm.org/10.1145/2767386.2767419.

[7] Peter Bailis and Kyle Kingsbury. "The network is reliable." In: *Queue* 12.7 (2014), p. 20.

[8] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. "Bolt-on causal consistency." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 761–772.

[9] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. "PRACTI replication." In: *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. San Jose, CA: USENIX Association, 2006, pp. 5–5.

[10] Kenneth P. Birman and Thomas A. Joseph. "Reliable communication in the presence of failures." In: *ACM Trans. Comput. Syst.* 5.1 (Jan. 1987), pp. 47–76. ISSN: 0734-2071. DOI: 10.1145/7351.7478.

[11] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors." In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[12] Eric A. Brewer. "Towards robust distributed systems (abstract)." In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. PODC '00. Portland, Oregon, United States: ACM, 2000, pp. 7–. ISBN: 1-58113-183-6. DOI: http://doi.acm.org/10.1145/343477.343502. URL: http://doi.acm.org/10.1145/343477.343502.

[13] Bernadette Charron-Bost. "Concerning the size of logical clocks in distributed systems." In: *Information Processing Letters* 39 (1991), pp. 11–16.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store." In: *SIGOPS Oper. Syst. Rev.* 41 (6 2007), pp. 205–220. ISSN: 0163-5980. DOI: http://doi.

acm.org/10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[16] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic algorithms for replicated database maintenance." In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM. 1987, pp. 1–12.

[17] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. "Orbe: Scalable causal consistency using dependency matrices and physical clocks." In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 11.

[18] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. "Gentlerain: Cheap and scalable causal consistency with physical clocks." In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–13.

[19] Colin Fidge. "Timestamps in Message-Passing Systems that preserve the Partial Ordering." In: *11th Australian Computer Science Conference*. 1989, pp. 55–66.

[20] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *SIGACT News* 33 (2 2002), pp. 51–59. ISSN: 0163-5700. DOI: http://doi.acm.org/10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[21] Richard A. Golding. "A Weak-Consistency Architecture for Distributed Information Services." In: *Computing Systems* 5 (1992), pp. 5–4.

[22] Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. "Concise Server-Wide Causality Management for Eventually Consistent Data Stores." In: *Distributed Applications and Interoperable Systems*. Springer. 2015, p. 66.

[23] James N Gray. "Notes on data base operating systems." In: *Operating Systems*. Springer, 1978, pp. 393–481.

[24] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[26] Paul R. Johnson and Robert H. Thomas. *The maintenance of duplicate databases*. Internet Request for Comments RFC 677. Information Sciences Institute, 1976.

[27] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent hashing and random trees." In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[28] J. J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System." In: *Thirteenth ACM Symposium on Operating Systems Principles*. Vol. 25. Asilomar Conference Center, Pacific Grove, US, 1991, pp. 213–225.

[29] Rusty Klophaus. "Riak Core: building distributed applications without shared state." In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 14:1–14:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900176. URL: http://doi.acm.org/10.1145/1900160.1900176.

[30] Nico Kruber, Maik Lange, and Florian Schintke. "Approximate Hash-Based Set Reconciliation for Distributed Replica Repair." In: *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*. IEEE. 2015, pp. 166–175.

[31] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. "Providing High Availability Using Lazy Replication." In: *ACM*

Trans. Comput. Syst. 10.4 (Nov. 1992), pp. 360–391. ISSN: 0734-2071. DOI: 10.1145/138873.138877.

[32] Avinash Lakshman and Prashant Malik. "Cassandra: a structured storage system on a P2P network." In: *SPAA*. Ed. by Friedhelm Meyer auf der Heide and Michael A. Bender. ACM, 2009, p. 47. ISBN: 978-1-60558-606-9. URL: http://doi.acm.org/10.1145/1583991.1584009.

[33] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[34] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[35] Richard J Lipton and Jonathan S Sandberg. *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[36] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 2011, pp. 401–416.

[37] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. "Stronger semantics for low-latency geo-replicated storage." In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 313–328.

[38] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. "Depot: Cloud Storage with Minimal Trust." In: *OSDI 2010*. Oct. 2010.

[39] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. "Consistency, availability, and convergence." In: *University of Texas at Austin Tech Report* 11 (2011).

[40] Dahlia Malkhi, Lev Novik, and Chris Purcell. "P2P replica synchronization with vector sets." In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 68–74.

[41] Dahlia Malkhi and Douglas B. Terry. "Concise Version Vectors in WinFS." In: *DISC*. Ed. by Pierre Fraigniaud. Vol. 3724. Lecture Notes in Computer Science. Springer, 2005, pp. 339–353. ISBN: 3-540-29163-6.

[42] Friedemann Mattern. "Virtual Time and Global Clocks in Distributed Systems." In: *Workshop on Parallel and Distributed Algorithms*. 1989, pp. 215–226.

[43] Ralph C. Merkle. "A Certified Digital Signature." In: *Proceedings on Advances in Cryptology*. CRYPTO '89. Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1989, pp. 218–238. ISBN: 0-387-97317-6.

[44] George V. Neville-Neil. "Time is an Illusion Lunchtime Doubly So." In: *Commun. ACM* 59.1 (Dec. 2015), pp. 50–55. ISSN: 0001-0782. DOI: 10.1145/2814336. URL: http://doi.acm.org/10.1145/2814336.

[45] Scott Owens, Susmit Sarkar, and Peter Sewell. "A better x86 memory model: x86-TSO." In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 391–407.

[46] D. Stott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. "Detection of Mutual Inconsistency in Distributed Systems." In: *Transactions on Software Engineering* 9.3 (1983), pp. 240–246.

[47] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. "Flexible Update Propagation for Weakly Consistent Replication." In: *Sixteen ACM Symposium on Operating Systems Principles*. Saint Malo, France, Oct. 1997.

[48] Ravi Prakash and Mukesh Singhal. "Dependency sequences and hierarchical clocks: Efficient alternatives to vector clocks for mobile computing systems." In: *Wireless Networks* (1997). also presented in Mobicom96, pp. 349–360.

[49] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. "Brief announcement: Efficient Causality Tracking in Distributed Storage Systems with Dotted Version Vectors." In: *Proceedings of the 2012 ACM symposium on PODC*. Madeira, Portugal: ACM, 2012, pp. 335–336. ISBN: 978-1-4503-1450-3. DOI: 10.1145/2332432.2332497.

[50] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. "Cimbiosys: A platform for content-based partial replication." In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. 2009, pp. 261–276.

[51] David Ratner, Peter L. Reiher, and Gerald J. Popek. "Roam: A Scalable Replication System for Mobility." In: *MONET* 9.5 (2004), pp. 537–544.

[52] Michel Raynal and Mukesh Singhal. "Logical Time: Capturing Causality in Distributed Systems." In: *IEEE Computer* 30 (Feb. 1996), pp. 49–56.

[53] R. Schwarz and F. Mattern. "Detecting causal relationships in distributed computations: In search of the Holy Grail." In: *Distributed Computing* 3.7 (1994), pp. 149–174.

[54] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free replicated data types." In: *Stabilization, Safety, and Security in DS*. Springer, 2011, pp. 386–400.

[55] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. "Transactional storage for geo-replicated systems." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 385–400.

[56] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. "Session guarantees for weakly consistent replicated data." In: *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*. IEEE. 1994, pp. 140–149.

[57] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. "Managing update conflicts in Bayou, a weakly connected replicated storage system." In: *ACM SIGOPS Operating Systems Review*. Vol. 29. 5. ACM. 1995, pp. 172–182.

[58] F. J. Torres-Rojas and M. Ahamad. "Plausible clocks: constant size logical clocks for distributed systems." In: *Distributed Computing* 12.4 (1999), pp. 179–196.

[59] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale Cluster Management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 18:1–18:17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964. URL: http://doi.acm.org/10.1145/2741948.2741964.

[60] Werner Vogels. "Eventually consistent." In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[61] Weihan Wang and Cristiana Amza. "On Optimal Concurrency Control for Optimistic Replication." In: *Proc. ICDCS*. 2009, pp. 317–326.

[62] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. "Write fast, read in the past: Causal consistency for client-side applications." In: *Proceedings of the 16th Annual Middleware Conference*. ACM. 2015, pp. 75–87.