DottedDB: Anti-Entropy without Merkle Trees, Deletes without Tombstones

Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero and Vitor Fonte
Universidade do Minho
HASLab, INESC TEC
Braga, Portugal
{tome,psa,cbm,vff}@di.uminho.pt

Abstract—**To achieve high availability in the face of network partitions, many distributed databases adopt eventual consistency, allow temporary conflicts due to concurrent writes, and use some form of per-key logical clock to detect and resolve such conflicts. Furthermore, nodes synchronize periodically to ensure replica convergence in a process called anti-entropy, normally using Merkle Trees. We present the design of 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. We evaluate DottedDB against MerkleDB, an otherwise identical database, but using per-key logical clocks and Merkle Trees for anti-entropy, to 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.**

Keywords-**Distributed Databases; Causality; Logical Clocks; Anti-Entropy; Merkle Trees; Partial Replication;**

I. Introduction

Internet-scale distributed systems are typically put together as a mix of different applications and sub-systems [1], often combining different trade-offs with respect to choices of consistency and availability in the face of partitions [2], [3]. In the last decade, since the advent of Amazon's Dynamo architecture [4], a growing niche of applications have been formed around eventually consistency solutions [5] supported by designs derived from Dynamo. These put together a key-value store supporting data divergence and reconciliation, with background anti-entropy repair supported by Merkle trees. Some of these eventually consistent (EC) databases, such as Apache Cassandra [6] and Basho's Riak [7], are now important components in the systems run by Comcast, Uber, Best Buy, Netflix and the UK National Health Service, among many others [8], [9].

These EC databases 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 [10].

While Cassandra and Riak have in common a client API with simple *get/put/delete* operations, they differ significantly from the Dynamo design on conflicting writes to the same key. Dynamo detected and tagged concurrent versions with *version vectors* [11]; Cassandra simplified the design by using *wall-clock timestamps*; Riak made it more precise by using *dotted version vectors* [12], allowing a more efficient handling of large numbers of concurrent writes.

Recent developments in eventually consistent designs have tried to solve known limitations of these early approaches: addressing causal consistency [13], [14], [15], [16], [17]; transactional support [16], [18], [14]; consistency tuning [19], [20]; and automating data reconciliation [21].

In this paper we only consider per-object causal consistency, not whole system cross-object causal consistency. We focus on a different set of problems that have so far received less attention. We will show that they have a significant impact on the efficiency and correctness of Dynamo-like system designs.

A. Challenges

It's recognized that "despite their simple interface and data model, distributed key-value storage systems are complex internally" [22]. Some of the challenges that arise from this complexity include: accurate and scalable *conflict detection*; reducing metadata pollution due to frequent *node churn*; correct and efficient handling of *distributed deletes*; and cost efficient support of *anti-entropy repair* of data. Next, we cover them in detail.

Conflict Detection: There are two main approaches: 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 [23], they would still fail to capture the *causality* [24] 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 term *logical clock* to denote any non-physical clock (not only Lamport scalar clocks, but also, e.g., vector clocks). Version Vectors [11] would detect the update conflict above, since updates originated from the same object and diverged independently. Logical clocks can either have node-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. Therefore, logical clocks may induce significant metadata costs and require further optimization in order to compete with the space efficiency of wall-clock timestamps.

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 induced a removal of entries via a LRU policy. However, removing entries is not safe, as a general solution, and can lead to false conflicts: wrongly identifying two versions as causally concurrent, when one *happened-before* [24] the other, making the latter 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 a linear growth in space consumption, which over time leads to significant waste. It would be desirable to have an alternative that is not linear with the number of deleted keys, in particular for small payloads where the relative cost of keeping tombstones is higher.

Anti-Entropy Repair: The network is unreliable [25] 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 [26] running as a background task.

The most used anti-entropy protocol in distributed databases involves hashing objects into a Merkle Tree [27] 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.

B. Contributions

To address the challenges above, in this paper we present the *Node-wide Dot-based Clocks (NDC)* Framework, involving several logical clocks, distributed algorithms, and corresponding data-structures. It establishes and exploits a close relationship between a *node logical clock* and every *object logical clock*. The key idea of NDC is to summarize (and migrate-to over time), information from every object logical clock into the node logical clock, leading to a much lower metadata footprint per object.

The NDC algorithms that we present in this paper can now automatically remove all unnecessary causal information from all objects. They also synchronize the partial shared state between nodes via anti-entropy, with no need for Merkle Trees. This leads to much better efficiency and behavior under failures than current approaches.

We developed *DottedDB*, a Dynamo-like distributed key-value store implementing the NDC framework. We evaluated its performance against *MerkleDB*, our implementation of a distributed database otherwise identical to DottedDB, but using the previous state of the art solutions to conflict detection (per-key Dotted Version Vectors) and anti-entropy repair (Merkle Trees). To summarize, the main contributions of this paper are as follows:

- A general NDC framework that enables: (a) efficient causality tracking and conflict detection; (b) distributed deletes without tombstones; (c) an lightweight anti-entropy protocol;
- DottedDB: an open-source distributed database based on the NDC framework;
- Extensive evaluation in a precisely controlled experiment to measure the impact of the novel approach (in DottedDB), under a fair comparison with previous state of the art techniques (in MerkleDB).

II. System Overview

A. 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.

B. 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.

C. 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:

$$\begin{aligned}(\text{values, context}) &\leftarrow \textbf{get}\ (\text{key})\\ \text{bool} &\leftarrow \textbf{put}\ (\text{key, value, context})\\ \text{bool} &\leftarrow \textbf{delete}\ (\text{key, context})\end{aligned}$$

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 subsequent 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.

III. Background and Related Work

A. Anti-Entropy Protocols

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 [26], typically being run periodically between pairs of nodes. The two most common data structures used to repair replicated data are *Bloom Filters* [28] and *Merkle Trees* [27]. 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 [29].

1) 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 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.

Accuracy versus Metadata: There is a fundamental tradeoff 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 [29].

B. Managing Multiple Versions

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.

1) Physical Timestamps: Some systems like Cassandra [6] tag each version of an object with a physical timestamp and use a last-writer-wins policy. This is simple to implement, but even perfect timestamps cannot capture the true nature of concurrent client updates, leading to lost updates without the client noticing.

2) Per-Object Logical Clocks: Others use the notion of causality [24] 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. Dynamo [4] uses the classic Version Vectors [11], a map from node ids to counters, incremented by the corresponding server node when coordinating an update request. Riak [7] uses an improved logical clock, Dotted Version Vectors [12], which allow accurate representation of client-originated concurrency.

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 VV with more and more entries over time. The normal approach to overcome this problem is to prune older entries in the VV, but this breaks causality and introduces false concurrency into the system [12].

3) Node-Wide 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. [30] developed Lazy Replication with node logical clocks and a Lamport clock [24] per write, but the metadata compaction depends on loosely-synchronized clocks and the availability of client replicas.

Bayou [31] attaches writes with a Lamport clock and stores them in 3 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 [14] focus is on causal consistency [32] 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) [33] 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 [34] improve on CVV by placing an upper-bound on the size of object logical clocks, dividing the objects in sets that can be represented by a single compact version vector, instead of a single node logical clock.

Cimbiosys [35] 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.

Gonçalves et al. [36] have the most similar use of node-based logical clocks, but nodes only exchange locally coordinated objects, which prevents data consistency in the presence of failures; also, the system does not guarantee complete object causality removal and node churn is not well supported because old node ids pollute the node clock.

IV. 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:

1) 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*[1]. 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.

2) 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.

3) 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.

4) 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.

5) 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.

A. Notation

We use mostly standard notation for sets and maps, including set comprehension of the form $\{f(x) \mid x \in S\}$ or $\{x \in S \mid \mathit{Pred}(x)\}$. 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. We use $\lhd$ for domain subtraction; $s \lhd m$ is the map obtained by removing from m all pairs (k, v) with $k \in s$. The domain and range of a map m is denoted by $\mathsf{dom}(m)$ and $\mathsf{ran}(m)$, respectively, i.e., $\mathsf{dom}(m) = \{k \mid (k, v) \in m\}$ and $\mathsf{ran}(m) = \{v \mid (k, v) \in m\}$.

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. We also use $\mathcal{P}(s)$ for the power set of some set s. For convenience, when $k \notin \mathsf{dom}(m)$, then $m[k]$ returns the *bottom* value for that type; e.g., for some $m \doteq \mathbb{I} \hookrightarrow \mathbb{N}$, then $m(k)$ returns 0 for any unmapped key k.

B. The Node Clock

The node clock represents which update events this node has *seen*, directly (coordinated by itself) or transitively (re-

[1] The term dot comes from [12], denoting an isolated event “over” a version vector, as a diacritic dot is placed over an “ı” to form an “i”.

ceived 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: an *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.

C. 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 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. But all gaps can be totally filled with extra dots, producing the *extrinsic* [36] 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 biggest dot per node id, like a Version Vector, without sacrificing correctness.

D. 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 bigger than 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:

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

E. Serving Client Requests

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

Algorithm 1: Client API at Node i.

```
procedure GET(Key $k$, Int $quorum[= 1]$):
    $O := \emptyset$
    for $p \in \mathsf{replica\_nodes}(k)$ do
        remote $\mathsf{fetch}(k)@p$ upon $o$ => $O := O \cup \{o\}$
    await $\mathsf{size}(O) \geq quorum$
    $(vers, cc) := \mathsf{reduce}(O, \mathsf{merge})$
    return $(\mathsf{ran}(vers), cc)$

procedure PUT(Key $k$, Value $v$, CausalContext $cc$):
    $c := \mathsf{max}(\mathsf{NC}_i[i]) + 1$
    $ver := ((i, c), v)$
    $cc[i] := c$
    $o := \mathsf{update}(k, (\{ver\}, cc))$
    for $p \in \mathsf{replica\_nodes}(k)$ do
        remote $\mathsf{update}(k, o)@p$

procedure DELETE(Key $k$, CausalContext $cc$):
    $\mathsf{PUT}(k, \mathbf{null}, cc)$
```

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.

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.

Algorithm 2: Auxiliary Operations at Node i.

```
function fetch(Key $k$):
    return $\mathsf{fill}(k, \mathsf{ST}_i[k], \mathsf{NC}_i)$

procedure store(Key $k$, Object $o$):
    $(vers, cc) := \mathsf{strip}(o, \mathsf{NC}_i)$
    // remove object if no values and $cc$ is empty
    if $\{\, val \in \mathsf{ran}(vers) \mid val \neq \textbf{null} \,\} = \emptyset \wedge cc = \emptyset$ then
        $\mathsf{ST}_i := \{k\} ⩤ \mathsf{ST}_i$
    else $\mathsf{ST}_i[k] := (vers, cc)$
    for $(n, c) \in \mathsf{dom}(vers)$ do
        $\mathsf{NC}_i[n] := \mathsf{NC}_i[n] \cup \{c\}$
        $\mathsf{DKM}_i[(n, c)] := k$
    if $cc = \emptyset$ then $\mathsf{NSK}_i := \mathsf{NSK}_i \setminus \{k\}$
    else $\mathsf{NSK}_i := \mathsf{NSK}_i \cup \{k\}$

function merge(Object $(v_1, cc_1)$, Object $(v_2, cc_2)$):
    $v := v_1 \cap v_2$
    $v_1 := \{\, (dot, val) \in v_1 \mid dot \notin cc_2 \,\}$
    $v_2 := \{\, (dot, val) \in v_2 \mid dot \notin cc_1 \,\}$
    return $(v \cup v_1 \cup v_2, cc_1 \cup cc_2)$

function update(Key $k$, Object $o$):
    $o' := \mathsf{fetch}(k)$
    $o := \mathsf{merge}(o, o')$
    $\mathsf{store}(k, o)$
    return $o$

function strip(Object $(vers, cc)$, NodeClock $\mathsf{NC}$):
    for $n \in \mathsf{dom}(cc)$ do
        // function base returns the biggest counter $b$ in
           some collection $C$, where $\forall i \in [1, b].\, i \in C$
        if $cc[n] \leq \mathsf{base}(\mathsf{NC}[n])$ then $cc := \{n\} ⩤ cc$
    return $(vers, cc)$

function fill(Key $k$, Object $(vers, cc)$, NodeClock $\mathsf{NC}$):
    for $n \in \mathsf{replica\_nodes}(k)$ do
        $cc[n] := \mathsf{max}(cc[n], \mathsf{base}(\mathsf{NC}[n]))$
    return $(vers, cc)$
```

Algorithm 3: Background Tasks at Node i.

```
process strip_causality():
    loop forever
        for $k \in \mathsf{NSK}_i$ do $\mathsf{store}(k, \mathsf{ST}_i[k])$
        $\mathsf{sleep}(strip\_interval)$

process anti_entropy():
    loop forever
        // $\mathsf{peers}(i)$ returns all peers of $i$ and $i$ itself
        remote $\mathsf{sync\_clock}(i, \mathsf{NC}_i)$@$\mathsf{random}(\mathsf{peers}(i) \setminus \{i\})$
        $\mathsf{sleep}(sync\_interval)$

procedure sync_clock(NodeId $p$, NodeClock $\mathsf{NC}_p$):
    // get all keys from dots missing in the node $p$
    $K := \emptyset$
    for $n \in \mathsf{peers}(i) \cap \mathsf{peers}(p)$ do
        $K := K \cup \{\, \mathsf{DKM}_i[(n, c)] \mid c \in \mathsf{NC}_i[n] \wedge c \notin \mathsf{NC}_p[n] \,\}$
    // get the missing objects from keys replicated by $p$
    $O := \{\, (k, \mathsf{ST}_i[k]) \mid k \in K \wedge p \in \mathsf{replica\_nodes}(k) \,\}$
    remote $\mathsf{sync\_repair}(i, \mathsf{NC}_i, O)$@$p$

procedure sync_repair(NodeId $p$, NodeClock $\mathsf{NC}_p$,
                      KeyObject's $O$):
    // update local objects with the missing objects
    for $(k, o) \in O$ do $\mathsf{update}(k, \mathsf{fill}(k, o, \mathsf{NC}_p))$
    // merge $p$'s node clock entry to close gaps
    $\mathsf{NC}_i[p] := \mathsf{NC}_i[p] \cup \mathsf{NC}_p[p]$
    // update the WM with new $i$ and $p$ clocks
    for $n \in \mathsf{dom}(\mathsf{NC}_p) \cap \mathsf{peers}(i)$ do
        $\mathsf{WM}_i[p][n] := \mathsf{max}(\mathsf{WM}_i[p][n], \mathsf{base}(\mathsf{NC}_p[n]))$
    for $n \in \mathsf{dom}(\mathsf{NC}_i)$ do
        $\mathsf{WM}_i[i][n] := \mathsf{max}(\mathsf{WM}_i[i][n], \mathsf{base}(\mathsf{NC}_i[n]))$
    // remove entries known by all peers
    for $(n, c) \in \mathsf{dom}(\mathsf{DKM}_i)$ do
        if $\mathsf{min}(\{\, \mathsf{WM}_i[m][n] \mid m \in \mathsf{peers}(n) \,\}) \geq c$ then
            $\mathsf{DKM}_i := \{(n, c)\} ⩤ \mathsf{DKM}_i$
```

F. Auxiliary Operations

In addition to the fill and strip operations already discussed, there are four other operations defined in Algorithm 2: 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 bigger 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) 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 obsolete 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.

G. Background Tasks

Algorithm 3 describes two background processes running at every node: the *anti-entropy* and the *causality stripping*.

1) Anti-Entropy: Periodically, the anti-entropy process in a node A chooses a random peer B to sync with, and sends its node clock. Node B computes the differences between A's clock and its own, resulting in a set of dots missing from A that B can provide. The dot-key map is queried by B to find which keys correspond to those missing dots. B then sends to A only those objects that are also replicated by A. Every object is read directly from storage without being filled, to save bandwidth; they are later filled at node A.

Upon receiving the missing objects, A updates all local objects with the received information. Additionally, it: (a) merges B's node clock entry into A's node clock, closing any gap from dots of B not replicated by A; (b) updates the watermark with the base of A's and B's node clock; (c) removes any entry in the dot-key map, if the dot is known by all peers (the entry is not necessary anymore for anti-entropy).

2) Causality Strip: 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.

V. 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.

A. DottedDB

We implemented a distributed database named DottedDB[2], using the NDC framework both for anti-entropy and per-object causality. DottedDB was implemented in Erlang, using the distributed systems framework Riak Core [7]. 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 handle client requests sequentially. Vnodes are totally ordered in a ring and assigned to physical machines sequentially. Data is replicated with consistent hashing [37]: 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.

B. MerkleDB

To establish a baseline, we implemented MerkleDB[3] as an optimized version of the key features from the state of art distributed key-value store design in Riak 2.0 and later versions. For fair comparison, MerkleDB is 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.

C. Configuration

We ran both DottedDB and MerkleDB in 64 independent vnodes, spread over a cluster of 5 physical machines. An additional machine was used to run YCSB [38], 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.

D. Object Logical Clock

Both systems use a logical clock to track object causality: MerkleDB uses a Dotted Version Vector, while DottedDB uses the NDC framework. We evaluate the scalability of the logical clock per object, how fast DottedDB can strip the clock and how that translates to actual distributed deletes, which directly depend on stripping.

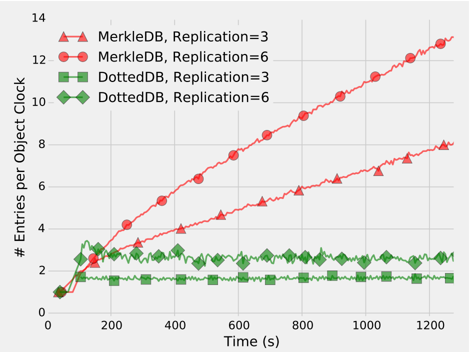


Fig. 1: Average number of entries in object clocks written to storage, for two different replication factors, with node churn.

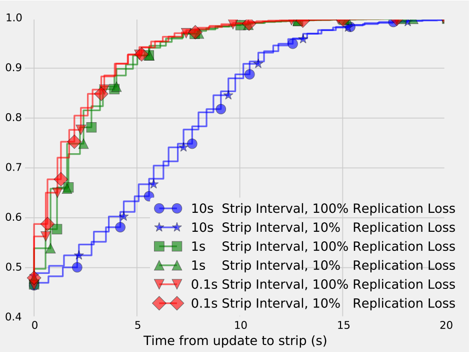


Fig. 2: CDF (Cumulative Distribution Function) of time needed to strip the causal past in an object's clock, after the update at the coordinating node.

1) Object Clock Scalability: Figure 1 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 measurement 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 1 clearly shows that the average size of the object clock keeps growing under node churn, as new updates add 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 average. This figure grows to an average of 3 entries per object for a replication factor of 6. Importantly, these numbers do not grow over time. New node ids from node churn do not pose a problem; entries are summarized by the node clock shortly after.

2) Strip Latency: Figure 2 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

[2] https://github.com/ricardobcl/DottedDB
[3] https://github.com/ricardobcl/MerkleDB

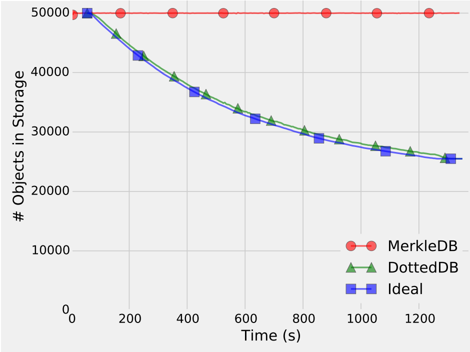


Fig. 3: Number of objects in storage over time. Initially 50 000 objects, serving 100 ops/s, 50% writes and 50% deletes.

(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% to simulate natural message loss, or 100% to simulate the extreme policy in which replica propagation is exclusively 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.

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 3 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 2 also represents the delay between a delete request and actual removal from storage.

E. Anti-Entropy

We compare the anti-entropy used by MerkleDB and DottedDB in three aspects: (1) *node metadata* used by each mechanism; (2) *network usage* while performing anti-entropy; (3) *replication latency*: the delay between an object being written by the coordinator and the object being stored in another replica.

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

TABLE I: 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%

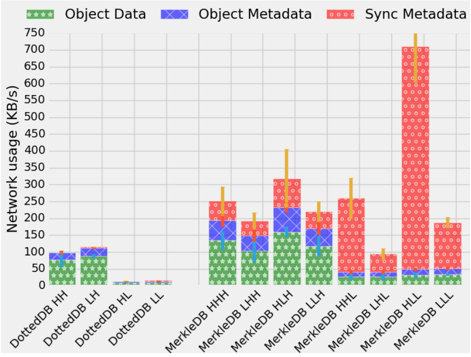


Fig. 4: Average network traffic used by the anti-entropy protocol.

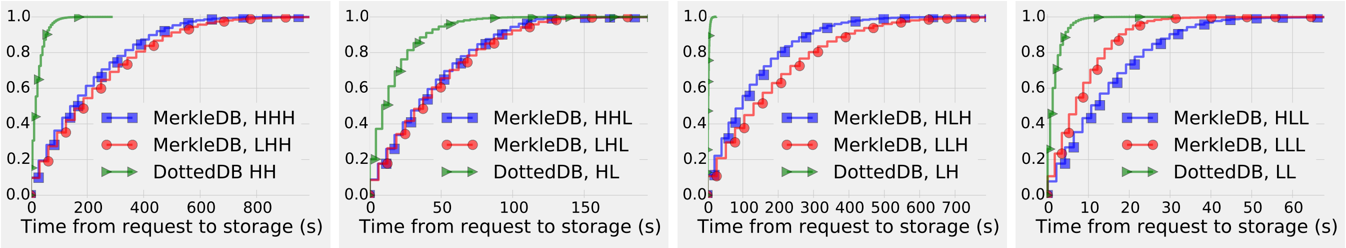
1) Metadata Size: MerkleDB uses one Merkle tree per replica group, due to partial replication, as explained in Section III-A1. For example, using consistent hashing and a replication factor of 3, each node has 3 Merkle trees. The space used per tree is linear with the number of keys (key-hash lists in leaf nodes), plus the size of the tree itself, which is fixed upon bootstrap.

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 and watermark are simply a version vector.

DottedDB metadata was very small (<10KB) when anti-entropy was frequent (LH and LL); even with infrequent syncs, metadata was still small (<2MB). In comparison, MerkleDB used at best 4.2MB. While metadata in DottedDB depends mostly on data divergence, which can be kept small by frequent syncs, in MerkleDB it depends on the number of keys per node, which grows with the database size.

2) Network Usage: MerkleDB sends each tree level in rounds, as necessary. If a leaf node is reached, all key-hashes in that leaf are sent, to compare hashes and see which objects are missing. DottedDB sends the node id, the node clock and missing objects (not filled).

Figure 4 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*



(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%).

Fig. 5: CDFs of the replication latency: the time from the moment a node coordinates an update, until the object is stored at another replica.

TABLE II: 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**
Average	19.8	16.4	**4.8**	5.1	7.0	9.3	6.7	8.1	6.4	8.9	7.0	8.0
$\mathbf{95^{th}}$	64	39	8	**7**	8	11	8	10	8	11	8	10
$\mathbf{99^{th}}$	460	394	**66**	79	137	173	128	155	123	163	143	149

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 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.

3) 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 times (the coordinator, always a replica node for the object, is excluded as its time would be 0). Figure 5 shows the CDFs for replication latency, where DottedDB is much faster at propagating data in every scenario. When replication message loss is 100% (replication exclusively by anti-entropy) and the sync interval is low (Figure 5c), DottedDB replicates 99% of updates in less than 20 seconds, while MerkleDB takes 20 times more.

F. 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. Table II 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.

VI. Conclusions

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. In this paper we show that consistent hashing and Merkle trees should not be used together; provide an alternative, based on tracking node-wide causality metadata; and demonstrate 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. In this paper we 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.

VII. Acknowledgments

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project «UID/EEA/50014/2013».

References

[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 18:1–18:17. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741964

[2] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, 2000.

[3] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007, pp. 205–220.

[5] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[7] R. Klophaus, "Riak core: building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010.

[8] A. S. Foundation, "Cassandra," http://cassandra.apache.org, accessed: 2016-10-20.

[9] Basho, "Riak," http://basho.com/about/customers/, accessed: 2016-10-20.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851

[11] D. S. Parker Jr, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, and D. Edwards, "Detection of mutual inconsistency in distributed systems," *IEEE Transactions on Software Engineering*, pp. 240–247, 1983.

[12] P. S. Almeida, C. Baquero, R. Gonçalves, N. Preguiça, and V. Fonte, "Scalable and accurate causality tracking for eventually consistent stores," in *Distributed Applications and Interoperable Systems*. Springer, 2014, pp. 67–81.

[13] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. 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.

[14] ——, "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.

[15] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 761–772.

[16] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. 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.

[17] S. Almeida, J. Leitão, and L. 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.

[18] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, 2013.

[19] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 265–278.

[20] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 6:1–6:16. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741972

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security in DS*. Springer, 2011, pp. 386–400.

[22] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and X. S. Li, "Eventually consistent: Not what you were expecting?" *Commun. ACM*, vol. 57, no. 3, pp. 38–44, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2576794

[23] G. V. Neville-Neil, "Time is an illusion lunchtime doubly so," *Commun. ACM*, vol. 59, no. 1, pp. 50–55, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2814336

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[25] P. Bailis and K. Kingsbury, "The network is reliable," *Queue*, vol. 12, no. 7, p. 20, 2014.

[26] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. 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.

[27] R. C. Merkle, "A certified digital signature," in *Proceedings on Advances in Cryptology*, ser. CRYPTO '89. New York, NY, USA: Springer-Verlag New York, Inc., 1989, pp. 218–238, http://dl.acm.org/citation.cfm?id=118209.118230.

[28] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[29] N. Kruber, M. Lange, and F. 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.

[30] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[31] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5. ACM, 1995, pp. 172–182.

[32] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[33] D. Malkhi and D. Terry, "Concise version vectors in winfs," in *Distributed Computing*. Springer, 2005, pp. 339–353.

[34] D. Malkhi, L. Novik, and C. Purcell, "P2p replica synchronization with vector sets," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 68–74, 2007.

[35] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. 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.

[36] R. Gonçalves, P. S. Almeida, C. Baquero, and V. Fonte, "Concise server-wide causality management for eventually consistent data stores," in *Distributed Applications and Interoperable Systems*. Springer, 2015, p. 66.

[37] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.