

Quantifying eventual consistency with PBS

Peter Bailis · Shivaram Venkataraman ·
Michael J. Franklin · Joseph M. Hellerstein ·
Ion Stoica

Received: 4 January 2013 / Revised: 6 June 2013 / Accepted: 8 July 2013 / Published online: 4 September 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract Data store replication results in a fundamental trade-off between operation latency and data consistency. At the weak end of the consistency spectrum is eventual consistency providing no limit to the staleness of data returned. However, anecdotally, eventual consistency is often “good enough” for practitioners given its latency and availability benefits. In this work, we explain why eventually consistent systems are regularly acceptable in practice, analyzing both the staleness of data they return and the latency benefits they offer. We introduce Probabilistically Bounded Staleness (PBS), a consistency model which provides expected bounds on data staleness with respect to both versions and wall clock time. We derive a closed-form solution for versioned staleness as well as model real-time staleness under Internet-scale production workloads for a large class of quorum-replicated, Dynamo-style stores. Using PBS, we measure the latency–consistency trade-off for partial, non-overlapping quorum systems, including limited multi-object operations. We quantitatively demonstrate how and why eventually consistent systems frequently return consistent data within tens of milliseconds while offering significant latency benefits.

Keywords Consistency · Replication · Quorum systems · Distributed databases · NoSQL · Staleness · Regular semantics · Dynamo · Prediction

All good ideas arrive by chance.—Max Ernst

1 Introduction

Modern distributed data stores need to be scalable, highly available, and fast. These systems typically replicate data across different machines and often across data centers for two reasons: first, to provide high availability when components fail and, second, to provide improved performance by serving requests from multiple replicas. In order to provide predictably low read and write latency, systems often eschew protocols guaranteeing “strong” consistency of reads and instead opt for eventually consistent protocols [3, 22, 27, 30, 45, 46, 63]. However, eventually consistent systems make no guarantees on the staleness of data items returned except that the system will “eventually” return the most recent version in the absence of new writes [74].

This latency–consistency trade-off inherent in distributed data stores has significant consequences for application design [3]. Low latency is critical for a large class of applications [67]. For example, at Amazon, 100 ms of additional latency resulted in a 1 % drop in sales [52], while 500 ms of additional latency in Google’s search resulted in a corresponding 20 % decrease in traffic [53]. At scale, increased latencies correspond to large amounts of lost revenue. However, lowering latency has a consistency cost: contacting fewer replicas for each request typically weakens the guarantees on returned data. Programs can often tolerate weak consistency by employing careful design patterns such as

P. Bailis (✉) · S. Venkataraman · M. J. Franklin · J. M. Hellerstein · I. Stoica

University of California, Berkeley, Berkeley, CA, USA
e-mail: pbailis@cs.berkeley.edu

S. Venkataraman
e-mail: shivaram@cs.berkeley.edu

M. J. Franklin
e-mail: franklin@cs.berkeley.edu

J. M. Hellerstein
e-mail: hellerstein@cs.berkeley.edu

I. Stoica
e-mail: istoica@cs.berkeley.edu

compensation (e.g., memories, guesses, and apologies) [40] and by using associative and commutative operations (e.g., timelines, logs, and notifications) [11]. However, potentially *unbounded* staleness (as in eventual consistency) poses significant challenges and is undesirable in practice.

1.1 Probabilistically Bounded Staleness

Distributed data store design allows a spectrum of consistency models, each requiring varying degrees of coordination. Modern stores frequently offer a choice between two modes of operation: “strong” consistency—often in the form of linearizability [24, 42], serializability [64], or regular register semantics [50]—and “weak” consistency, most often (but not necessarily [57, 72]) in the form of eventual consistency. Despite eventual consistency’s weak guarantees, data store operators frequently choose this option [1, 22, 30, 45, 63, 77]—a controversial decision [39, 54, 68, 69]. Given their performance benefits, which are especially important as latencies grow [3, 30, 39, 40], eventually consistent store configurations are often considered acceptable. The proliferation of eventually consistent deployments suggests that applications can often tolerate occasional staleness and that data tend to be “fresh enough” in many cases.

While common practice suggests that eventual consistency is often a viable solution for data store operators, to date, this observation has been largely anecdotal. In this work, we quantify the degree to which eventual consistency is both eventual and consistent and explain why. Under worst-case conditions, eventual consistency results in an unbounded degree of data staleness, but, as we will show, the average case is often different. Eventually consistent data stores cannot promise strong consistency but, for varying degrees of certainty, can offer staleness bounds with respect to time (“how eventual”) and versions (“how consistent”).

There is little prior work describing how to make these consistency and staleness predictions under practical conditions. The current state of the art requires that users make rough guesses or perform online profiling to determine the consistency provided by their data stores [19, 35, 75]. Users have little to no guidance on how to choose an appropriate replication configuration or how to predict the behavior of eventual consistency in production environments.

To predict consistency, we need to know when and why eventually consistent systems return stale data and how to quantify the staleness of the data they return. In this paper, we present algorithms and models for predicting the staleness, called Probabilistically Bounded Staleness (PBS). There are two common metrics for measuring staleness in the literature: wall clock time [35, 73, 78, 79] and versions [35, 48, 81]. PBS describes both measures, providing the probability of reading a write Δ seconds after it returns ((Δ, p) -semantics, or “how eventual is eventual consistency?”), of reading one of the

last K versions of a data item ((K, p) -semantics, or “how consistent is eventual consistency?”), and of experiencing a combination of the two ((K, Δ, p) -semantics). PBS does not propose new mechanisms to enforce deterministic staleness bounds [48, 62, 78, 79, 81]; instead, our goal is to provide a lens for analyzing, improving, and predicting the behavior of *existing*, widely deployed systems.

1.2 Practical partial quorums

In this work, we use PBS to examine the latency–consistency trade-off in the context of quorum-replicated data stores such as Dynamo [27] and its open source descendants Apache Cassandra [49], Basho Riak [13], and Project Voldemort [31]. Quorum systems ensure strong consistency across reads and writes to replicas by ensuring that read and write replica sets overlap. However, employing *partial* (or non-strict) quorums can lower latency by requiring fewer replicas to respond. With partial quorums, sets of replicas written to and read from need not overlap: given N replicas and read and write quorum sizes R and W , partial quorums imply $R + W \leq N$.

We expand prior work on *probabilistic quorums* [58, 60] to account for multi-version staleness and messaging protocols as used in today’s systems. We derive closed-form solutions for PBS (K, p) -regular semantics and use Monte Carlo methods to explore the trade-off between latency and (Δ, p) -regular semantics. Using production latency distributions, we present a detailed study of Dynamo-style PBS (Δ, p) -regular semantics. We show how long-tailed one-way write latency distributions affect the time required for a high probability of consistent reads. For example, in one production environment, switching from spinning disks to solid-state drives dramatically improved consistency (e.g., 1.85 vs. 45.5 ms wait time for a 99.9% probability of consistent reads) due to decreased write latency mean and variance. We also make quantitative observations of the latency–consistency trade-offs offered by partial quorums. For example, in another production environment, we observe an 81.1% combined read and write latency improvement at the 99.9th percentile (230–43.3 ms) for a 202-ms window of inconsistency (99.9% probability consistent reads). This analysis demonstrates the performance benefits that lead operators to choose eventual consistency.

We make the following contributions in this paper:

- We develop the theory of PBS for partial quorums. PBS can describe the probability of staleness across versions ((K, p) -semantics) and time ((Δ, p) -semantics) as well as the probability of session-based consistency.
- We provide a closed-form analysis of (K, p) -regular semantics for quorum systems and demonstrate how the probability of receiving data k versions old is exponentially reduced by k . As a corollary, (K, p) -regular seman-

tics tolerance also exponentially lowers quorum system load.

- We describe the WARS model for (Δ, p) -regular semantics in Dynamo-style partial quorum systems and show how message reordering leads to staleness. We evaluate the (Δ, p) -regular semantics of Dynamo-style systems using a combination of synthetic and production latency models.
- We present theoretical and empirical analysis for the likelihood of two kinds of multi-key operations: transactional atomicity and causal consistency. We evaluate the probability of causal consistency using real-world workloads and also describe our experiences in integrating the PBS predictor in production data stores.

This paper is an invited, extended version of “Probabilistically Bounded Staleness for Practical Partial Quorums,” which appeared in VLDB 2012 [15] and extends this prior work in several ways. We have added analysis of multi-key guarantees including causality and atomicity (Sect. 7) and have extended our discussion of PBS design and implementation, including consistency prediction versus verification, white-box versus black-box modeling, and our experiences with real-world stores (Sect. 8). We have completely revised our definitions of PBS metrics (Sect. 2) so they are more comparable with related work and have fixed several errata from the prior published version of this work. While this resulting article is substantially longer than our initial paper, we believe it provides a more comprehensive treatment of the material.

The remainder of this paper is organized as follows: in Sect. 2, we define PBS metrics for general distributed storage systems. In Sect. 3, we provide background on quorum systems in theory and practice, which we will study with PBS in Sect. 4. In Sect. 5, we develop PBS metrics for a specific class of quorum systems, patterned on Amazon’s Dynamo, which we use in Sect. 6 to quantitatively analyze the behavior of data stores deployed in production. In Sect. 7, we discuss two kinds of multi-key guarantees: transactional atomicity and causal consistency. We present a discussion on PBS design and implementation in Sect. 8, including the differences between white-box and black-box PBS techniques, prediction versus verification, and our experiences integrating PBS in production data stores. In Sect. 9, we describe related work, and, in Sect. 10, we conclude.

2 Probabilistically Bounded Staleness

In this work, we develop quantitative metrics for describing the behavior of eventually consistent systems. To do so, we first introduce several general-purpose metrics that will guide the remainder of our work, which applies them to

quorum-replicated stores in real-world deployments. Metrics presented in this section are applicable to any data store.

Quantifying eventual consistency is difficult because eventual consistency is a particularly weak guarantee. According to the popular definition by Werner Vogels, a system is eventually consistent if it “guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value” [74]. Under this definition, eventual consistency is purely a liveness guarantee—something good eventually happens—but it does not provide any safety properties—anything can happen in the meantime [10]. There is no specification of *how long* it takes until the last value is returned or *what happens* when we attempt to read a data item and the latest value is not returned. If a system makes *guarantees* about these properties, then it will provide a stronger consistency model than Vogel’s basic eventual consistency above.¹ In this work, we consider the problem of quantifying the consistency of a system that does not provide additional guarantees beyond basic eventual consistency (i.e., replica convergence).

This lack of safety guarantees does not preclude us from making quantitative statements about the behavior of eventually consistent stores. In this work, we take a probabilistic approach to consistency that allows us to reason about the *expected* consistency of an eventually consistent store. In our probabilistic metrics, we draw on two safety guarantees from the literature: time- and version-based staleness. While these properties have been studied in *deterministic* contexts, we extend their semantics to account for probabilistic behavior.² Collectively, we term these metrics PBS.

As a side note, we must be careful in describing a store as providing a particular semantics with a given probability. For example, if a store provides a given semantics with 50% probability, does this mean that half of the read operations will obey the semantics or that only half of the store *deployments* will obey the semantics? Both are admissible, but the former is likely preferable. Even considering read-based probabilistic semantics, distribution of violations across operations may vary: a system providing 500 “consistent” reads followed by 500 “inconsistent” reads is, according to our tentative definition, equivalent to a system that alternates between the two response types. In this work, we do

¹ Many systems providing consistency semantics such as serializability, linearizability, and convergent causal consistency provide eventual consistency (often along with *additional* safety and liveness properties which result in stronger models than “basic” eventual consistency).

² In prior versions of this work [15], we used different terminology; here, we leverage definitions from the existing literature. We believe these metrics will both provide greater clarity and allow for cleaner integration with other metrics. For readers familiar with our prior terminology, k -staleness has become (K, p) -regular semantics, t -visibility has become (Δ, p) -regular semantics, and (k, t) -staleness has become (Δ, K, p) -regular semantics.

not distinguish between the two cases but, in practice, coupling these probabilistic guarantees with windowed semantics (e.g., “over S seconds,” “over W successive writes”) will disallow the above undesirable behavior.

2.1 System model

We consider a standard model for a data storage system. The system contains a collection of data items which clients write to and read from. Each write creates a new version of a data item, and reads return a previously written version (or set of versions) of the requested item. The semantics of the data store determine which versions of a given data item can be returned in response to reads. This model is consistent with the original work from which its deterministic counterparts are derived below.

2.2 PBS versioned staleness

To begin, we consider version-based staleness. Aiyer et al. [8] expanded Lamport’s *atomic* (linearizable—not transactionally atomic), *regular*, and *safe* distributed register semantics [50] to account for multi-versioned distributed registers. We reproduce their definition of K -atomic semantics below, adapting their definition to a read-centric (as opposed to system-centric) consistency model:

Definition 1 A read obeys K -atomic semantics if there exists an order of the operations that is consistent with real-time order and one of the versions returned by the read is equal to one of the versions written by the last K preceding writes in the order (assuming there are K initial writes with the same initial value) [8].

K -atomic semantics provide safety guarantees on reads, which are substantially more complex than only providing basic eventual consistency (in fact, this difficulty was the subject of the work from Aiyer et al. [8,9]). However, we can adapt K -atomic semantics to a probabilistic context:

Definition 2 A system provides (K, p) -atomic semantics if reads obey K -atomic semantics with probability p .

Note that PBS (K, p) -atomic semantics effectively have two variables: probability and versions. In practice, when we consider data store behavior, we often wish to explore the Pareto frontier between them, or, in the context of predictions or SLAs, treat one as dependent and the other as independent.

We can also consider K -regular semantics (also from Aiyer et al.), which weaken atomic semantics to relax constraints on in-flight writes. Under *regular* semantics, if a read to an item does not overlap with a write, it must return the effects of the prior write. If the read overlaps with a write (that is, a write [or set of writes] starts, then a read is issued and completes before the write[s] complete), then the read

can return the value of any of the in-flight writes or the prior version of the write [50]. As illustrated in Fig. 2a, K -regular semantics allow a write to return either in-flight writes or any of the last K completed writes:

Definition 3 A read obeys K -regular semantics if, in the case that the read that does not overlap with a write, it returns the result of one of the latest K completed writes, or, if the read overlaps with a write, it returns either the result of one of the latest K completed writes or the eventual result of one of the overlapping writes [8].

Definition 4 A system provides (K, p) -regular semantics if reads obey K -regular semantics with probability p .

We omit a similar translation for safe semantics.

Using version-based (K, p) semantics, we can predict whether a client will ever read older data than it has previously read, a well-known session guarantee called *monotonic reads consistency* [72]. This is particularly useful when clients do not need to see the most recent version of a data item but still require a notion of “forward progress” through versions.

Definition 5 A system obeys p -monotonic reads consistency if, with probability p , at least one value in any read quorum returned to a client is the same version or a newer version than the last version that the client previously read.

To guarantee that a client sees monotonically increasing versions, it can continue to contact the same replica [74] (provided the “sticky” replica does not fail). However, this is insufficient for strict monotonic reads, where the client reads strictly newer data if it exists in the system. We can adapt Definition 5 to accommodate strict monotonic reads by requiring that the data store returns a more recent data version if it exists.

2.3 PBS time staleness

Consistency is also often expressed in terms of wall clock time [73,78,79]. Recent work by Golab et al. considered the problem of verifying time-based staleness in a theoretical context [35]. We adopt their terminology for deterministic atomicity and refer the interested reader to their analysis for a full case-by-case analysis of (deterministic) atomicity violations:

Definition 6 A read obeys Δ -atomic semantics if it returns the latest write or the value of the latest write as of up to Δ time units ago [35].

We can similarly adapt the definition of Δ -atomic semantics to a probabilistic context:

Definition 7 A system provides (Δ, p) -atomic semantics if reads obey Δ -atomic semantics with probability p .

We can adapt the definition of (Δ, p) -atomic semantics to a Δ -regular register and, subsequently, a PBS Δ -regular register:

Definition 8 A read obeys Δ -regular semantics if, in the case that it does not overlap with a write, it returns either the latest write or the value of the latest write as of up to Δ time units ago, or, if the read overlaps with a write, it returns either the latest write, the value of the latest write as of up to Δ time units ago, or the eventual result of one of the overlapping writes.

Definition 9 A system provides (Δ, p) -regular semantics if reads obey Δ -regular semantics with probability p .

We again omit a translation for safe semantics.

2.4 PBS time and versions

We can combine the previous models to combine both versioned and real-time staleness metrics. We can describe the conjunction of both time- and version-based consistency properties; as an example, consider (K, Δ) -atomicity and (K, Δ, p) -atomicity:

Definition 10 A read obeys (K, Δ) -atomic semantics if there exists an order of the operations that is consistent with real-time order and one of the versions returned by the read is equal to one of the versions written by the last K preceding writes in the order as of up to Δ time units ago (assuming there are K initial writes with the same initial value).

Definition 11 A system provides (K, Δ, p) -regular semantics if reads obey (K, Δ) -regular semantics with probability p .

Note that (K, Δ, p) -atomic semantics encapsulate the prior definitions of consistency. (K, p) -atomic semantics are equivalent to $(K, \Delta = 0, p)$ -atomic semantics, while (Δ, p) -atomic semantics are equivalent to $(K = 1, \Delta, p)$ -atomic semantics.

As above, we can repeat this exercise for regular and safe semantics.

2.5 Using PBS metrics

PBS metrics are independent of the underlying data store implementation. Intuitively, if we have information about the relevant underlying behavior and current operating conditions of an eventually consistent store (e.g., replication protocols, network latency), then we can quantify their effects on the store's semantics. For certain architectures, like master-slave systems, this is relatively straightforward (e.g., the transit time from master to slave—often already measured—is

easily correlated with staleness). For others, like the quorum systems we will study, this is more complicated. Without this white-box information, we can still perform rough black-box modeling or monitoring. We discuss these alternatives in Sect. 8.1 and several prior approaches for consistency enforcement (i.e., not prediction) in Sect. 9.

3 Quorum system background

While PBS metrics provide a foundation for quantifying eventually consistent system behavior, they are more useful when applied to a given data store. In this work, we consider PBS metrics in the context of quorum-replicated data stores, which represent a large class of widely deployed real-world distributed data stores.

In this section, we provide background on quorum systems both in the theoretical academic literature and in practice. We begin by introducing prior work on traditional and probabilistic quorum systems. We next discuss Dynamo-style quorums, currently the most widely deployed protocol for data stores employing quorum replication. Finally, we survey reports of practitioner usage of partial quorums for three Dynamo-style data stores.

3.1 Quorum foundations: theory

Systems designers have long proposed quorum systems as a replication strategy for distributed data [34]. Under quorum replication, a data store writes a data item by sending it to a set of replicas, called a write quorum. To serve reads, the data store fetches the data from a possibly different set of replicas, called a read quorum. For reads, the data store compares the set of values returned by the replicas, and, given a total ordering of versions, can return the most recent value (or all values received, if desired). For each operation, the data store chooses read and write quorums from a set of sets of replicas, known as a *quorum system*, with one system per data item. There are many kinds of quorum systems, but one simple configuration is to use read and write quorums of fixed sizes, which we will denote R and W , for a set of nodes of size N . To reiterate, a quorum-replicated data store uses one quorum system per data item. Across data items, quorum systems need not be identical. Finally, the minimum sized quorum defines the system's fault tolerance, or availability.

Informally, a strict quorum system is a quorum system with the property that any two quorums in the quorum system overlap (have non-empty intersection). Reading and writing to R and W replicas in a strict quorum provides regular semantics. A simple example of a strict quorum system is the majority quorum system, in which each quorum is of size $\lceil \frac{N+1}{2} \rceil$. The theory literature describes alternative quorum system designs providing varying asymptotic

properties of capacity, scalability, and fault tolerance, from tree-quorums [5] to grid-quorums [61] and highly available hybrids [6]. Jiménez-Peris et al. [44] provide an overview of traditional, strict quorum systems. Importantly, note that using overlapping read and write sets does not guarantee linearizability. Instead, overlapping read and write quorums provides regular semantics [50], and, accordingly, we will focus on regular semantics in this work.

Partial quorum systems are natural extensions of strict quorum systems: at least two quorums in a partial quorum system do not overlap. There are two relevant variants of partial quorum systems described in the literature: probabilistic quorum systems and k -quorums.

Probabilistic quorum systems provide probabilistic guarantees of quorum intersection. By scaling the number of replicas, one can achieve an arbitrarily high probability of consistency [58]. Intuitively, this is a consequence of the Birthday Paradox: as the number of replicas increases, the probability of non-intersection between any two quorums decreases. Probabilistic quorums are typically used to predict the probability of strong consistency but not (multi-version) bounded staleness. Merideth and Reiter provide an overview of these systems [60].

As an example of a probabilistic quorum system, consider N replicas with randomly chosen read and write quorums of sizes R and W . We can calculate the probability that the read quorum does not contain the last written version. This probability is the number of quorums of size R composed of nodes that were not written to in the write quorum divided by the number of possible read quorums:

$$p_s = \frac{\binom{N-W}{R}}{\binom{N}{R}} \quad (1)$$

The probability of inconsistency is high except for large N . With $N = 100$, $R = W = 30$, $p_s = 1.88 \times 10^{-6}$ [8]. However, with $N = 3$, $R = W = 1$, $p_s = 0.6$. The asymptotics of these systems are excellent—but only asymptotically.

k-quorum systems provide *deterministic* guarantees that a partial quorum system will return values that are within k versions of the most recent write [8]. In a single writer scenario, sending each write to $\lceil \frac{N}{k} \rceil$ replicas with round-robin write scheduling ensures that any replica is no more than k versions out-of-date. However, with multiple writers, we lose the global ordering properties that the single writer was able to control, and the best-known algorithm for the pathological case results in a lower bound of $(2N - 1)(k - 1) + N$ versions staleness [9].

This prior work makes two important assumptions. First, it typically models quorum sizes as fixed, where the set of nodes with a version does not grow over time. Prior work examined “dynamic systems,” considering quorum membership churn [4], network-aware quorum placement [32,36],

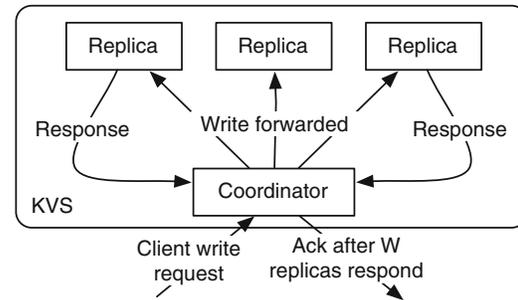


Fig. 1 Diagram of control flow for client write to Dynamo-style quorum ($N = 3$, $W = 2$). A coordinator node handles the client write and sends it to all N replicas. The write call returns after the coordinator receives W acknowledgments

and network partitions [41] but not write propagation. Second, it frequently assumes Byzantine failure. We revisit these assumptions in the next section.

3.2 Quorum foundations: practice

In practice, many distributed data management systems use quorums as a replication mechanism. Amazon’s Dynamo [27] is the progenitor of a class of eventually consistent key-value stores that include Apache Cassandra [49], Basho Riak [13], and LinkedIn’s Project Voldemort [31]. All of these systems use the same variant of quorum-style replication and we are not aware of any widely adopted data store using a vastly different quorum replication protocol. However, with some work, we believe that other styles of replication can adopt our methodology. We describe key-value stores here, but any replicated data store can use quorums, including traditional RDBMS systems.

Dynamo-style quorum systems employ one quorum system per key, typically maintaining the mapping of keys to quorum systems using a consistent-hashing scheme or a centralized membership protocol. Each node in the system cluster stores multiple keys. As shown in Fig. 1, clients send read and write requests to a node in the system cluster, which forwards the request to *all* nodes assigned to that key as replicas. This coordinating node considers an operation complete when it has received responses from a pre-determined number of replicas (typically set per-operation). Accordingly, without message loss, all replicas eventually receive all writes. This means that the write and read quorums chosen for a request depend on which nodes respond to the request first. Dynamo denotes the replication factor of a key as N , the number of replica responses required for a successful read as R , and the number of replica acknowledgments required for a successful write as W . Under normal operation, Dynamo-style systems guarantee regular semantics when $R + W > N$. While there is ongoing work on providing atomic (linearizable) semantics in these stores [21], we are not aware of any

stores that currently provide atomic semantics. Accordingly, to analyze the likelihood that eventually consistent configurations behave like strongly consistent configurations, we focus on regular semantics.

There are significant differences between quorum theory and data systems used in practice. First, replication factors for data stores are low, typically between one and three [22, 30, 37]. Second (in the absence of failure), in Dynamo-style partial quorums, the write quorum size increases even after the operation returns, growing via anti-entropy [28]. Coordinators send all requests to all replicas but consider only the first R (W) responses. As a matter of nomenclature (and to disambiguate against “dynamic” quorum membership protocols), we will refer to these systems as *expanding partial quorum systems*. (We discuss additional anti-entropy in Sect. 5.2.) Third, as in much of the applied literature, practitioners focus on fail-stop instead of Byzantine failure modes [20]. Following standard practice, we do not consider Byzantine failure.

3.3 Typical quorum configurations

For improved latency, operators often set $R + W \leq N$. Here, we survey quorum configurations according to practitioner accounts. Operators frequently use partial quorum configurations, citing performance benefits and high availability. Most of these accounts did not discuss the possibility or occurrence of staleness resulting from partial quorum configurations.

Cassandra defaults to $N = 3, R = W = 1$ [22]. The Apache Cassandra 1.0 documentation claims that “a majority of users do writes at consistency level [$W = 1$],” while the Cassandra Query Language defaults to $R = W = 1$ as well [1]. Production Cassandra users report using $R = W = 1$ in the “general case” because it provides “maximum performance” [77], which appears to be a commonly held belief [45, 63]. Cassandra has a “minor” patch [2] for session guarantees [72] that is not currently used [29]; according to our discussions with developers, this is due to lack of interest.

Riak defaults to $N = 3, R = W = 2$ [17, 18]. Users suggest using $R = W = 1, N = 2$ for “low value” data (and strict quorum variants for “web,” “mission critical,” and “financial” data) [46, 55].

Finally, Voldemort does not provide sample configurations, but Voldemort’s authors (and operators) at LinkedIn [30] often choose $N = c, R = W = \lceil c/2 \rceil$ for odd c . For applications requiring “very low latency and high availability,” LinkedIn deploys Voldemort with $N = 3, R = W = 1$. For other applications, LinkedIn deploys Voldemort with $N = 2, R = W = 1$, providing “some consistency,” particularly when three-way replication is not required. Unlike Dynamo, Voldemort sends read requests to R of N replicas (not N of N) [31]; this decreases load per replica and network traffic at the expense of read latency and potential availability. Provided staleness probabilities are independent across

requests, this does not affect staleness: even when sending reads to N replicas, coordinators only wait for R responses.

4 PBS and quorums

In this section, we use PBS metrics introduced in Sect. 2 to analyze the consistency of modern quorum systems. We discuss (K, p) -regular semantics first because they are self-contained, with a simple closed-form solution. In comparison, analyzing (Δ, p) -regular semantics is more difficult, involving additional variables. Accordingly, this section proceeds in order of increasing difficulty, and much of the remainder of this paper addresses the complexities of (Δ, p) -regular semantics.

Practical concerns guide the following theoretical contributions. We begin by considering a model without quorum expansion or other anti-entropy. For the purposes of a running example, as in Eq. 1, we assume that W (R) of N replicas are randomly selected for each write (read) operation. Similarly, we consider fixed W, R and N across multiple operations. Next, we expand our model to consider write propagation in expanding partial quorums. In this section, we discuss anti-entropy in general; however, we model Dynamo-style quorums in Sect. 5. We discuss further refinements to these assumptions in Sect. 8.

4.1 PBS (K, p) -regular semantics

Probabilistic quorums allow us to determine the probability of returning the most recent value written to the database, but do not describe what happens when the most recent value is not returned. Here, we determine the probability of returning a value within a bounded number of versions. In the following formulation, we consider traditional, non-expanding write quorums (no anti-entropy). Reads may return versions whose writes that are not yet committed (in-flight) (see Fig. 2a).

The probability of returning a version of a key within the last k versions committed is equivalent to intersecting one of k independent write quorums. Given the probability of a single quorum non-intersection p , the probability of non-intersection with one of the last k independent quorums is p^k . In our running example, the probability of non-intersection is Eq. 1 exponentiated by k :

$$p = \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right)^k \quad (2)$$

When $N = 3, R = W = 1$, this means that the probability of returning a version within 2 versions is 0.5 , within 3 versions, 0.703 , 5 versions, >0.868 , and 10 versions, >0.98 . When $N = 3, R = 1, W = 2$ (or, equivalently, $R = 2, W =$

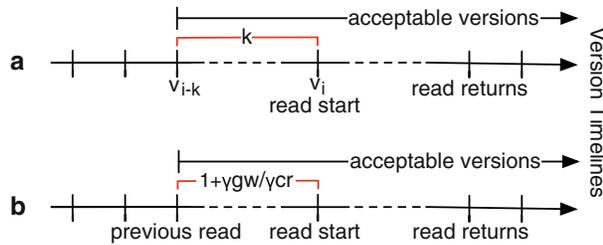


Fig. 2 Valid versions returned by read operations under PBS (K, p) -regular semantics (a) and PBS monotonic reads (b). Each timeline represents the linear, real-time order of versions according to write completion time. In Dynamo-style systems, the correct version(s) to return typically corresponds to the highest-timestamped write(s). In (K, p) -regular semantics, the read operation will return a version no later than k versions older than the last committed value when it started. In monotonic reads consistency, acceptable staleness depends on the number of versions committed since the client's last read

1), these probabilities increase: $k = 1 \rightarrow 0.6$, $k = 2 \rightarrow 0.8$, and $k = 5 \rightarrow > 0.995$.

This closed-form solution holds for quorums that do not change size over time. For expanding partial quorum systems, this solution is a bound on p for (K, p) -regular semantics.

4.2 p -Monotonic reads consistency

p -Monotonic reads consistency is a special case of PBS (K, p) -regular semantics (see Fig. 2b), where k is determined by a client's rate of reads from a data item (γ_{cr}) and the global, system-wide rate of writes to the same data item (γ_{gw}). If we know these rates, the number of versions written between client reads is $\frac{\gamma_{gw}}{\gamma_{cr}}$, as shown in Fig. 2b. We can calculate the probability of monotonic reads as a special case of (K, p) -regular semantics where $k = 1 + \frac{\gamma_{gw}}{\gamma_{cr}}$. Again extending our running example, from Eq. 2:

$$p = \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right)^{1 + \gamma_{gw}/\gamma_{cr}} \quad (3)$$

For strict monotonic reads, where we cannot read the version we have previously read (assuming there are newer versions in the database), we exponentiate with $k = \frac{\gamma_{gw}}{\gamma_{cr}}$.

In practice, we may not know these exact rates, but, by measuring their distribution, we can calculate an expected value. By performing appropriate admission control, operators can control these rates to achieve monotonic reads consistency with high probability.

4.3 Load improvements with version staleness

The *load* of a quorum system is defined as the frequency of accessing the busiest quorum member [61, Definition 3.2]. Intuitively, the busiest quorum member limits the number of

requests that a given quorum system can sustain, called its *capacity* [61, Corollary 3.9].

Prior work determined that probabilistic quorum systems did not offer significant benefits to load (providing a constant factor improvement compared to strict quorum systems) [58]. Here, we show that quorums tolerating PBS (K, p) -regular semantics have asymptotically lower load than traditional probabilistic quorum systems (and, transitively, than strict quorum systems).

The probabilistic quorum literature defines an ε -intersecting quorum system as a quorum system that provides a $1 - \varepsilon$ probability of returning consistent data [58, Definition 3.1]. A ε -intersecting quorum system has load of at least $\frac{1 - \sqrt{\varepsilon}}{\sqrt{N}}$ [58, Corollary 3.12].

In considering k versions of staleness, we consider the intersection of k ε -intersecting quorum systems. For a given probability p of inconsistency, if we are willing to tolerate k versions of staleness, we need only require that $\varepsilon = \sqrt[k]{p}$. This implies that our PBS (K, p) -regular semantics system construction has load of at least $\frac{1 - p^{\frac{1}{2k}}}{\sqrt{N}}$, an improved lower bound compared to traditional probabilistic quorum systems. PBS monotonic reads consistency results in a lower bound on load of $\frac{1 - p^{\frac{1}{2C}}}{\sqrt{N}}$, where $C = 1 + \frac{\gamma_{gw}}{\gamma_{cr}}$.

These results are intuitive: if we are willing to tolerate multiple versions of staleness, we need to contact fewer replicas. Staleness tolerance lowers the load of a quorum system, subsequently increasing its capacity.

4.4 PBS (Δ, p) -regular semantics

Until now, we have considered only quorums that do not grow over time. However, as we discussed in Sect. 3.2, real-world quorum systems expand by asynchronously propagating writes to quorum system members over time. This process is commonly known as anti-entropy [28]. In this section, we will restrict our discussion to generic anti-entropy techniques. However, we explicitly model the Dynamo-style anti-entropy mechanisms in Sect. 5.

PBS (Δ, p) -regular semantics models the probability of inconsistency for expanding quorums. This Δ captures the expected length of the “window of inconsistency.” Recall that we consider in-flight writes—which are more recent than the last committed version—as non-stale.

In our analysis, we consider the worst case for Δ : we analyze (Δ, p) -regular semantics as if the last write happened exactly Δ seconds ago. This is for two reasons. First, this captures a notion of “visibility,” or how long it takes a write to appear to readers. This formulation is interesting because it considers the period of maximum vulnerability to inconsistency. For example, (Δ, p) -regular semantics are likely guaranteed (high p) when the last write completed, say, seconds

or minutes (or longer) ago but are much lower immediately after the write completes. Second, this formulation allows us our analysis to be independent of read and write rates. If the last write happened Δ^* seconds ago, choosing any $\Delta < \Delta^*$ will be a conservative estimate. With multiple overlapping writes, the probability of consistency similarly increases.

We denote the probability density function describing the probability that *exactly* \mathcal{W}_r replicas have received a particular version v , Δ seconds after v commits as $P_w(\mathcal{W}_r, \Delta)$.

By definition, for expanding quorums, $\forall c \in [0, W]$, $P_w(c, 0) = 1$; at commit time, W replicas will have received the value with certainty. We can model the probability of PBS (Δ, p) -regular semantics for given Δ by summing the conditional probabilities of each possible \mathcal{W}_r :

$$p = \sum_{c \in [W, N]} \frac{\binom{N-c}{R}}{\binom{N}{R}} \cdot [P_w(c, \Delta) - P_w(c + 1, \Delta)] \tag{4}$$

However, the above equation assumes reads occur instantaneously and writes commit immediately after W replicas have the version (i.e., there is no delay acknowledging the write to the coordinating node). In the real world, coordinators wait for write acknowledgments and read requests take time to arrive at remote replicas, increasing Δ . Accordingly, Eq. 4 is a conservative upper bound on p .

In practice, P_w depends on the anti-entropy mechanisms in use and the expected latency of operations, but we can approximate it (Sect. 5) or measure it online. For this reason, the load of a (Δ, p) quorum system depends on write propagation and is difficult to analytically determine for general-purpose expanding quorums. Additionally, one can model both transient and permanent failures by increasing the tail probabilities of P_w (Sect. 6.9).

4.5 PBS (K, Δ, p) -regular semantics

We can extend our prior analyses of (K, p) -regular and (Δ, p) -regular semantics to consider (K, Δ, p) -regular semantics:

$$p = \left(\sum_{c \in [W, N]} \frac{\binom{N-c}{R}}{\binom{N}{R}} \cdot [P_w(c, \Delta) - P_w(c + 1, \Delta)] \right)^K \tag{5}$$

In this equation, in addition to (again) assuming instantaneous reads, we also assume the pathological case where the last k writes all occurred at the same time. If we can determine the time since commit for the last k writes, we can improve this bound by considering each quorum's p separately (individual Δ). However, predicting (and enforcing) write arrival rates is challenging and may introduce inaccuracy, so this equation is a conservative lower bound on p .

In practice, we believe it is easier to reason about staleness of versions or staleness of time but not both together.

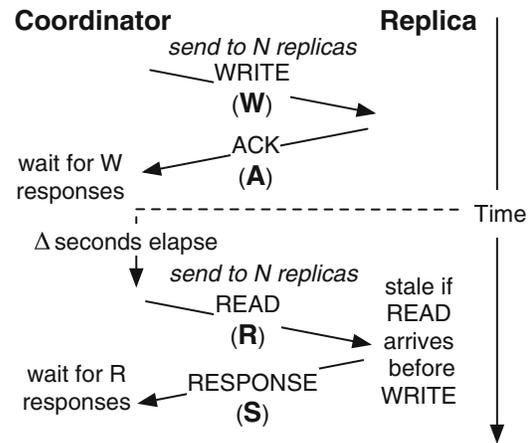


Fig. 3 The WARS model for in Dynamo describes the message latencies between a coordinator and a single replica for a write followed by a read t seconds after commit. In an N replica system, this messaging occurs N times

Accordingly, having derived a closed-form model for (K, p) -regular semantics, in the remainder of this paper, we focus mainly on deriving more specific models for (Δ, p) -regular semantics. A conservative rule-of-thumb going forward is to exponentiate the probability of inconsistency in (Δ, p) -regular semantics by k when up to k versions of staleness are tolerable.

5 Dynamo-style (Δ, p) -regular semantics

We have a closed-form model for (K, p) -regular semantics, but (Δ, p) -regular semantics are dependent on both the quorum replication algorithm and the anti-entropy processes employed by a given system. In this section, we discuss PBS (Δ, p) -regular semantics in the context of Dynamo-style data stores and describe how to asynchronously detect staleness.

5.1 Inconsistency in Dynamo: WARS model

Dynamo-style quorum systems are inconsistent as a result of read and write message reordering, in turn a product of message delays. To illustrate this phenomenon, we introduce a model of message latency in Dynamo operation that, for convenience, we call WARS.

In Fig. 3, we illustrate WARS using a space–time diagram for messages between a coordinator and a single replica for a write followed by a read Δ seconds after the write commits. Δ here corresponds to the Δ in PBS (Δ, p) -regular semantics. In brief, reads are stale when all of the first R responses to the read request arrived at their replicas before the last (committed) write request.

For a write, the coordinator sends N messages, one to each replica. The message from the coordinator to replica contain-

ing the write is delayed by a value drawn from distribution \bar{w} . The coordinator waits for W responses from the replicas before it can consider the version committed. Each response acknowledging the write is delayed by a value drawn from the distribution A .

For a read, the coordinator (possibly different than the write's coordinator, and possibly representing a different client than the client that issued the write) sends N messages, one to each replica. The message from coordinator to replica containing the read request is delayed by a value drawn from distribution R . The coordinator waits for R responses from the replicas before returning the most recent value it receives. The read response from each replica is delayed by a value drawn from the distribution S .

The read coordinator will return stale data if the first R responses received reached their replicas before the replicas received the latest version (delayed by \bar{w}). When $R + W > N$, this is impossible. However, under partial quorums, the frequency of this occurrence depends on the latency distributions. If we denote the commit time (when the coordinator has received W acknowledgments) as w_t , a single replica's response is stale if $r' + w_t + \Delta < w'$ for r' drawn from R and w' drawn from \bar{w} . Writes have time to propagate to additional replicas both while the coordinator waits for all required acknowledgments (A) and as replicas wait for read requests (R). Read responses are further delayed in transit (S) back to the read coordinator, inducing further possibility of reordering. Qualitatively, longer write tails (\bar{w}) and faster reads increase the chance of staleness due to reordering.

WARS considers the effect of message sending, delays, and reception, but this represents a daunting analytical formulation. The commit time is an order statistic of W and N dependent on both \bar{w} and A . Furthermore, the probability that the i th returned read message observes reordering is another order statistic of R and N dependent on \bar{w}, A, R , and S . Moreover, across responses, the probabilities are dependent. These dependencies make calculating the probability of staleness rather difficult. Dynamo is straightforward to reason about and program but is difficult to analyze in a simple closed form. As we discuss in Sect. 6.1, we instead explore *WARS* using Monte Carlo methods, which are straightforward to understand and implement.

5.2 *WARS* scope

5.2.1 Proxying operations

Depending on which coordinator a client contacts, coordinators may serve reads and writes locally. In this case, subject to local query processing delays, a read or write to R or W nodes behaves like a read or write to $R - 1$ or $W - 1$ nodes. Although we do not do so, one could adopt *WARS* to handle local reads and writes. The decision to proxy requests (and,

if not, which replicas serve which requests) is data store and deployment specific. Dynamo forwards write requests to a designated coordinator solely for the purpose of establishing a version ordering [27, Section 6.4] (easily achievable through other mechanisms [43]). Dynamo's authors observed a latency improvement by proxying all operations and having clients act as coordinators—Voldemort adopts this architecture [70].

5.2.2 Client-side delays

End-users will likely incur additional time between their reads and writes due to latency required to contact the service. Individuals making requests to Web services through their browsers will likely space sequential requests by tens or hundreds of milliseconds due to client-to-server latency. Although we do not consider this delay here, it is important to remember for practical scenarios because delays between reads and writes (Δ) for individual clients may be large.

5.2.3 Additional anti-entropy

As we discussed in Sect. 3.2, anti-entropy decreases the probability of staleness by propagating writes between replicas. Dynamo-style systems also support additional anti-entropy processes [59]. *Read repair* is a commonly used anti-entropy process: when a read coordinator receives multiple versions of a data item from different replicas in response to a read request, it will attempt to (asynchronously) update the out-of-date replicas with the most recent version [27, Section 5]. Read repair acts like an additional write for every read, except old values are re-written. Additionally, Dynamo used Merkle trees to summarize and exchange data contents between replicas [27, Section 4.7]. However, not all Dynamo-style data stores actively employ similar gossip-based anti-entropy. For example, Cassandra uses Merkle tree anti-entropy only when manually requested (e.g., `nodetool repair`), instead relying primarily on quorum expansion and read repair [23].

These processes are rate dependent: read repair's efficiency depends on the rate of reads, and Merkle tree exchange's efficiency (and, more generally, most anti-entropy efficiency) depends on the rate of exchange. A conservative assumption for read repair and Merkle tree exchange is that they never occur. For example, assuming a particular read repair rate implies a given rate of reads from each key in the system.

In contrast, *WARS* captures expanding quorum behavior independent of read rate and with conservative write rate assumptions. *WARS* considers a single read and a single write. Aside from load considerations, concurrent reads do not affect staleness. If multiple writes overlap (that is, have overlapping periods where they are in-flight but are not committed), the probability of inconsistency decreases. This is

because overlapping writes result in an increased chance that a client reads as-yet-uncommitted data. As a result, with WARS, data may be fresher than predicted.

5.3 Asynchronous staleness detection

Even if a system provides a low probability of inconsistency, applications may need notification when data returned is inconsistent or staler than expected. Here, as a side note, we discuss how the Dynamo protocol is naturally equipped for staleness detection. We focus on PBS (Δ, p) -regular semantics in the following discussion, but it is easily extended to other PBS semantics.

Knowing whether a response is stale at read time requires strong consistency. Intuitively, by checking all possible values in the domain against a hypothetical staleness detector, we could determine the (strongly) consistent value to return. While we cannot do so synchronously, we *can* determine staleness asynchronously. Asynchronous staleness detection allows speculative execution [76] if a program contains appropriate compensation logic.

We first consider a staleness detector providing false positives. Recall that, in a Dynamo-style system, we wait for R of N replies before returning a value. The remaining $N - R$ replicas will still reply to the read coordinator. Instead of dropping these messages, the coordinator can compare them to the version it returned. If there is a mismatch, then either *i.*) the coordinator returned stale data, *ii.*) there are in-flight writes in the system, or *iii.*) additional versions committed after the read. The latter two cases, relating to data committed after the response initiation, lead to false positives. In these cases, the read did not return “stale” data even though there were newer but uncommitted versions in the system. Notifying clients about newer but uncommitted versions of a data item is not necessarily bad but may be unnecessary. This detector does not require modifications to the Dynamo protocol and is similar to the read-repair process.

To eliminate these uncommitted-but-newer false positives (cases two and three), we need to determine the total, system-wide commit ordering of writes. Recall that replicas are unaware of the commit time for each version. The timestamps stored by replicas are not updated after commit, and commits occur after W replicas respond. Thankfully, establishing a total ordering among distributed agents is a well-known problem that a Dynamo-style system can solve by using a centralized service [43] or using distributed consensus [51]. This requires modifications but is feasible.

6 Evaluating Dynamo (Δ, p) -regular semantics

As discussed in Sect. 4.4, PBS (Δ, p) -regular semantics depends on the propagation of reads and writes throughout

a system. We introduced the WARS model as a means of reasoning about inconsistency in Dynamo-style quorum systems, but quantitative metrics such as staleness observed in practice depend on each of WARS’s latency distributions. In this section, we perform an analysis of Dynamo-style (Δ, p) -regular semantics to better understand how frequently “eventually consistent” means “consistent” and, more importantly, why.

PBS (K, p) -regular semantics is easily captured in closed form (Sect. 4.1). It does not depend on write latency or any environmental variables. Indeed, in practice, without expanding quorums or anti-entropy, we observe that our derived equations hold true experimentally.

(Δ, p) -regular semantics depends on anti-entropy, which is more complicated. In this section, we focus on deriving experimental expectations for PBS (Δ, p) -regular semantics. While we could improve the staleness results by considering additional anti-entropy processes (Sect. 5.2), we make the minimum of assumptions required by the WARS model. Conservative analysis decreases the number of experimental variables (supported by empirical observations from practitioners) and increases the applicability of our results.

6.1 Monte Carlo simulation

In light of the complicated analytical formulation discussed in Sect. 5.1, we implemented WARS analysis in a Monte Carlo based simulation. Calculating (Δ, p) -regular semantics for a given value of Δ is straightforward (Algorithm 1). To simulate the message delays between coordinator and each of the N replicas, denote the i th sample drawn from distribution D as $D[i]$ and draw N samples from $W, A, R,$ and S (lines 4–14). Compute the time of the write request completes (w_t , or the time that the coordinator gets its W th acknowledgment; the W th smallest value of $\{w[i] + A[i], i \in [0, N]\}$, calculated in lines 9 and 15). Next, determine whether any of the first R replicas contained an up-to-date response: check whether any the first R samples of R , ordered by $R[i] + S[i]$ (lines 16–22) obey $w_t + R[i] + \Delta \leq W[i]$ (lines 23–28). Repeating this process multiple times (lines 3–29) provides an approximation of the behavior specified by the trace (line 30). Extending this formulation to analyze (K, Δ, p) -regular semantics given a distribution of write arrival times will require accounting for multiple writes across time.

6.2 Experimental validation

To validate WARS, our simulation implementation, and our subsequent analyses, we compared our predicted (Δ, p) -regular semantics and latency with the consistency we observed in a commercially supported, open source Dynamo-style data store. We modified Cassandra to profile WARS latencies, disabled read repair (as it is external to WARS),

Algorithm 1 Calculating p under (Δ, p) -regular semantics using WARS model.

```

1: given:  $N, R, W, \Delta$ , WARS model, iterations
2: consistent_trials = 0
3: for  $i = 1 \rightarrow \text{iterations}$  do
   (generate WARS latencies for each replica
   to find write and read latencies)
4:  $Ws = \{\}; As = \{\}; Rs = \{\}; Ss = \{\};$ 
5: write_latencies =  $\{\};$  read_latencies =  $\{\};$ 
6: for replica = 0  $\rightarrow N$  do
7:    $Ws[\text{replica}] = \text{WARS.nextW}();$ 
8:    $As[\text{replica}] = \text{WARS.nextA}();$ 
9:   write_latencies[replica] =  $Ws[\text{replica}] + As[\text{replica}];$ 
10:
11:    $R_s[\text{replica}] = \text{WARS.nextR}();$ 
12:    $S_s[\text{replica}] = \text{WARS.nextS}();$ 
13:   read_latencies[replica] =  $R_s[\text{replica}] + S_s[\text{replica}];$ 
14: end for
   (the  $W$ th fastest reply determines write
   finish)
15: write_finish = find_nth_smallest_element(write_latencies,  $W$ );
   (the  $R$ th fastest reply determines read
   finish)
16: read_finish = find_nth_smallest_element(read_latencies,  $R$ );
   (find the first  $R$  replicas that replied)
17: reply_replicas =  $\{\};$ 
18: for replica = 0  $\rightarrow N$  do
19:   if read_latencies[replica]  $\leq$  read_finish then
20:     reply_replicas.append(replica)
21:   end if
22: end for
   (determine if any responses were
   consistent)
23: for replica  $\in$  reply_replicas do
24:   if write_finish +  $R_s[\text{replica}] + \Delta \geq Ws[\text{replica}]$  then
25:     consistent_trials += 1;
26:     break;
27:   end if
28: end for
29: end for
30: return consistent_trials/iterations;

```

and, for reads, only considered the first R responses (often, more than R messages would arrive by the processing stage, decreasing staleness). We ran Cassandra on three servers with 2.2 GHz AMD Opteron 2214 dual-core SMT processors and 4 GB of 667 MHz DDR2 memory, serving in-memory data. To measure staleness, we inserted increasing versions of a key while concurrently issuing read requests and performed post-hoc log analysis to determine observed consistency.

Our WARS predictions matched our empirical observations of Cassandra's behavior. We injected a combination of exponentially distributed latencies into Cassandra; this was necessary to observe non-negligible inconsistency in our test cluster and approximates the long-tailed behavior of the production latency distributions we explore later (Sect. 6.4). In general, these exponential distributions had substantially higher variance than those we saw in practice

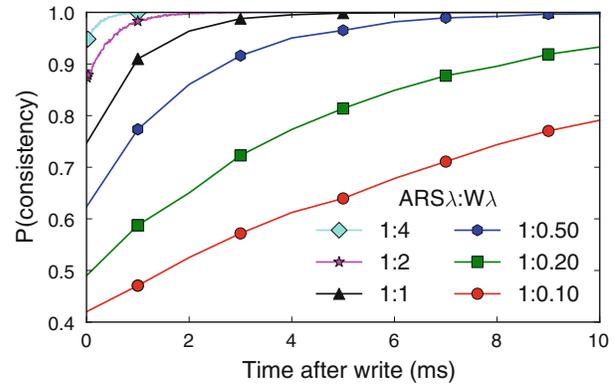


Fig. 4 (Δ, p) -regular semantics with exponential latency distributions for W and $A=R=S$. Mean latency is $1/\lambda$. $N = 3, R = W = 1$

(which fit well with Pareto bodies and exponential tails). We injected each combination of exponentially distributed $W = \lambda \in \{0.05, 0.1, 0.2\}$ (means 20, 10, and 5 ms) and $A = R = S = \lambda \in \{0.1, 0.2, 0.5\}$ (means 10, 5, and 2 ms) across 50,000 writes. After empirically measuring the WARS distributions, consistency, and latency for each partial quorum configuration, we predicted the (Δ, p) -regular semantics and latency. Our average (Δ, p) -regular semantics prediction RMSE was 0.28 % (std. dev. 0.05 %, max. 0.53 %) for each $t \in \{1, \dots, 199\}$ ms. Our predicted latency (for each of the $\{1.0, \dots, 99.9\}$ th percentiles for each configuration) had an average N-RMSE of 0.48 % (std. dev. 0.18 %, max. 0.90 %). This validates our Monte Carlo based implementation for IID distributions.

6.3 Write latency distribution effects

As discussed in Sect. 5.1, the WARS model of Dynamo-style systems dictates that high one-way write variance (W) increases staleness. To quantify these effects, we swept a range of exponentially distributed write distributions (changing parameter λ , which dictates the mean and tail of the distribution) while fixing $A=R=S$.

Our results, shown in Fig. 4, confirm this relationship. When the variance of W is 0.0625 ms ($\lambda = 4$, mean .25 ms, one-fourth the mean of $A=R=S$), we observe a 94 % chance of consistency immediately after the write and 99.9 % chance after 1 ms. However, when the variance of W is 100 ms ($\lambda = .1$, mean 10 ms, ten times the mean of $A=R=S$), we observe a 41 % chance of consistency immediately after write and a 99.9 % chance of consistency only after 65 ms. As the variance and mean increase, so does the probability of inconsistency. Under distributions with fixed means and variable variances (uniform, normal), we observe that the mean of W is less important than its variance if W is strictly greater than $A=R=S$.

Table 1 LinkedIn Voldemort single-node production latencies

Percentile	Latency (ms)
15,000 RPM SAS Disk	
Average	4.85
95	15
99	25
Commodity SSD	
Average	0.58
95	1
99	2

Decreasing the mean and variance of W improves the probability of consistent reads. This means that, as we will see, techniques that lower one-way write latency result in higher chance of consistent reads. Instead of increasing read and write quorum sizes, operators could choose to lower (relative) W latencies through hardware configuration or by delaying reads. This latter option is potentially detrimental to performance for read-dominated workloads and may introduce undesirable queuing effects.

6.4 Production latency distributions

To study *WARS* in greater detail, we obtained production latency statistics from two Internet-scale companies.

LinkedIn³ is an online professional social network with over 225 million members as of July 2013. To provide highly available, low latency data storage, engineers at LinkedIn built Voldemort. Alex Feinberg, a lead engineer on Voldemort, graciously provided us with latency distributions for a single node under peak traffic for a user-facing service at LinkedIn, representing 60% read and 40% read-modify-write traffic [30] (Table 1). Feinberg reports that, using spinning disks, Voldemort is “largely IO bound and latency is largely determined by the kind of disks we’re using, [the] data to memory ratio and request distribution.” With solid-state drives (SSDs), Voldemort is “CPU and/or network bound (depending on value size).” As an aside, Feinberg also noted that “maximum latency is generally determined by [garbage collection] activity (rare, but happens occasionally) and is within hundreds of milliseconds.”

Yammer⁴ provides private social networking to over 200,000 companies as of July 2013 and uses Basho’s Riak for some client data [13]. Coda Hale, an infrastructure architect, and Ryan Kennedy, also of Yammer, previously presented in-depth performance and configuration details for their Riak deployment in March 2011 [38]. Hale provided us

Table 2 Yammer Riak $N = 3, R = 2, W = 2$ production latencies

Percentile	Read latency (ms)	Write latency (ms)
Min	1.55	1.68
50	3.75	5.73
75	4.17	6.50
95	5.2	8.48
98	6.045	10.36
99	6.59	131.73
99.9	32.89	435.83
Max	2,979.85	4,465.28
Mean	9.23	8.62
Standard deviation	83.93	26.10
Mean rate	718.18 gets/s	45.65 puts/s

with more detailed performance statistics for their application [37] (Table 2). Hale mentioned that “reads and writes have radically different expected latencies, especially for Riak.” Riak delays writes “until the fsync returns, so while reads are often < 1 ms, writes rarely are.” Also, although we do not model this explicitly, Hale also noted that the size of values is important, claiming “a big performance improvement by adding LZFP compression to values.”

6.5 Latency model fitting

While the provided production latency distributions are invaluable, they are under-specified for *WARS*. First, the data are summary statistics, but *WARS* requires distributions. More importantly, the provided latencies are round-trip times, while *WARS* requires the constituent one-way latencies for both reads and writes. As our validation demonstrated, these latency distributions are easily collected, but, because they are not currently collected in production, we must fill in the gaps. Accordingly, to fit $W, A, R,$ and S for each configuration, we made a series of assumptions. Without additional data on the latency required to read multiple replicas, we assume that each latency distribution is independently, identically distributed (IID). We fit each configuration using a mixture model with two distributions, one for the body and the other for the tail.

LinkedIn provided two latency distributions, whose fits we denote LNKD-SSD and LNKD-DISK for the SSD and spinning disk data. As previously discussed, when running on SSDs, Voldemort is network and CPU bound. Accordingly, for LNKD-SSD, we assumed that read and write operations took equivalent amounts of time and, to allocate the remaining time, we focused on the network-bound case and assumed that one-way messages were symmetric ($W=A=R=S$). Feinberg reported that Voldemort performs at least one read

³ LinkedIn. www.linkedin.com.

⁴ Yammer. www.yammer.com.

Table 3 Distribution fits for production latency distributions from LinkedIn (LNKD-*) and Yammer (YMMR)

LNKD-SSD	$W = A = R = S :$ 91.22%: Pareto, $x_m = .235, \alpha = 10$ 8.78%: Exponential, $\lambda = 1.66$ N-RMSE: .55%
LNKD-DISK	$W:$ 38%: Pareto, $x_m = 1.05, \alpha = 1.51$ 62%: Exponential, $\lambda = .183$ N-RMSE: .26% $A = R = S : \text{LNKD-SSD}$
YMMR	$W:$ 93.9%: Pareto, $x_m = 3, \alpha = 3.35$ 6.1%: Exponential, $\lambda = .0028$ N-RMSE: 1.84% $A = R = S :$ 98.2%: Pareto, $x_m = 1.5, \alpha = 3.8$ 1.8%: Exponential, $\lambda = .0217$ N-RMSE: .06%

before every write (average of 1 seek, between 1 and 3 seeks), and writes to the BerkeleyDB Java Edition back-end flush to durable storage either every 30 s or 20 MB—whichever comes first [30]. Accordingly, for LNKD-DISK, we used the same $A=R=S$ as LNKD-SSD but fit W separately.

Yammer provided distributions for a single configuration, denoted YMMR, but separated read and write latencies. Under our IID assumptions, we fit single-node latency distributions to the provided data, again assuming symmetric $A, R,$ and S . The data again fit a Pareto distribution with a long exponential tail. At the 98th percentile, the write distribution takes a sharp turn. Fitting the data closely resulted in a long tail, with 99.99+th percentile writes requiring tens of seconds—much higher than Yammer specified. Accordingly, we fit the 98th percentile knee conservatively; without the 98th percentile, the write fit N-RMSE is .104%.

We also considered a wide-area network replication scenario, denoted WAN. Reads and writes originate in a random data center, and, accordingly, one replica command completes quickly and the coordinator routes the others remotely. We delay remote messages by 75 ms and apply LNKD-DISK delays once the command reaches a remote data center, reflecting multi-continent WAN delay [26].

We show the parameters for each distribution in Table 3 and plot each fitted distribution in Fig. 5. Note that for R, W of one, LNKD-DISK is not equivalent to WAN. In LNKD-DISK, we only have to wait for one of N local reads (writes) to return, whereas, in WAN, there is only one local read (write) and the network delays all other read (write) requests by at least 150 ms.

6.6 Production (Δ, p) -regular semantics

We measured the (Δ, p) -regular semantics for each distribution (Fig. 6). As we observed under synthetic distributions in Sect. 6.3, the (Δ, p) -regular semantics depended on both the relative mean and variance of W .

LNKD-SSD and LNKD-DISK demonstrate the importance of write latency in practice. Immediately after write commit, LNKD-SSD had a 97.4% probability of consistent reads, reaching over a 99.999% probability of consistent reads after 5 ms. LNKD-SSD's reads briefly raced with its writes immediately after commit. However, within a few milliseconds after the write, the chance of a read arriving before the last write was nearly eliminated. The distribution's read and write operation latencies were small (median .489 ms), and writes completed quickly across all replicas due to the distribution's short tail (99.9th percentile .657 ms). In contrast, under LNKD-DISK, writes take much longer (median 1.50 ms) and have a longer tail (99.9th percentile 10.47 ms). LNKD-DISK's (Δ, p) -regular semantics reflects this difference: immediately after write commit, LNKD-DISK had only a 43.9% probability of consistent reads and, 10 ms later, only a 92.5% probability. This suggests that SSDs may greatly improve consistency due to reduced write variance.

We experienced similar effects with the other distributions. Immediately after commit, YMMR had a 89.3% chance of consistency. However, YMMR's long tail hampered its (Δ, p) -regular semantics and only reached a 99.9% probability of consistency 1,364 ms after commit. As expected, WAN observed poor chances of consistency until after the 75 ms passed (33% chance immediately after commit); the client had to wait longer to observe the most recent write unless it originated from the reading client's data.

6.7 Quorum sizing

In addition to $N = 3$, we consider how varying the number of replicas (N) affects (Δ, p) -regular semantics while maintaining $R = W = 1$. The results, depicted in Fig. 7, show that the probability of consistency immediately after write commit decreases as N increases. With 2 replicas, LNKD-DISK has a 57.5% probability of consistent reads immediately after commit but only a 21.1% probability with 10 replicas. However, at high probabilities (p), the wait time required for increased replica sizes is close. For LNKD-DISK, the (Δ, p) -regular semantics at 99.9% probability of consistency ranges from 45.3 ms for 2 replicas to 53.7 ms for 10 replicas.

These results imply that maintaining a large number of replicas for availability or better performance results in a potentially large impact on consistency immediately after writing. However, the (Δ, p) -regular semantics probability (p) will still rise quickly (low Δ).

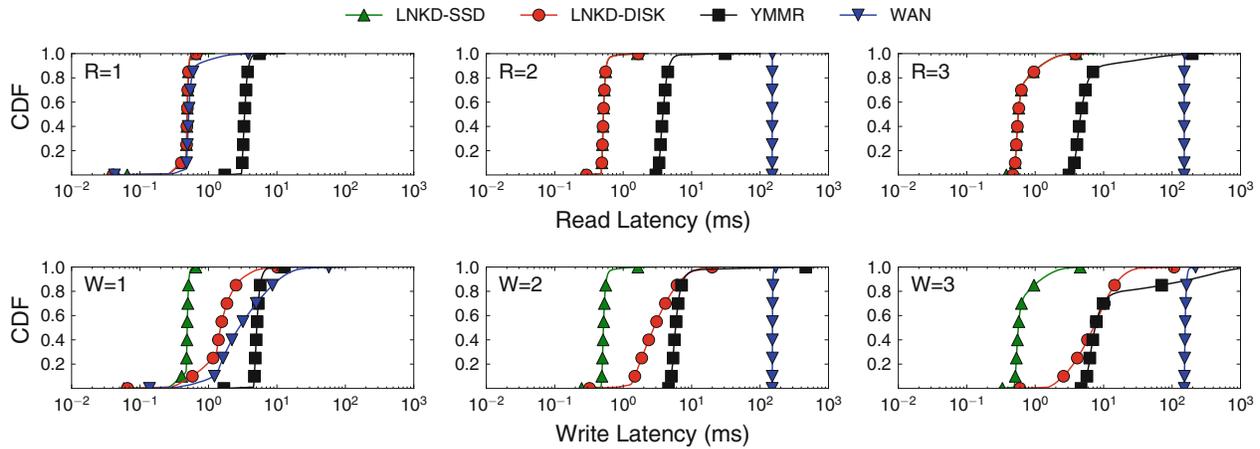


Fig. 5 Read and write operation latency for production fits for $N = 3$. For reads, LNKD-SSD is equivalent to LNKD-DISK. Higher values of R and W result in latency

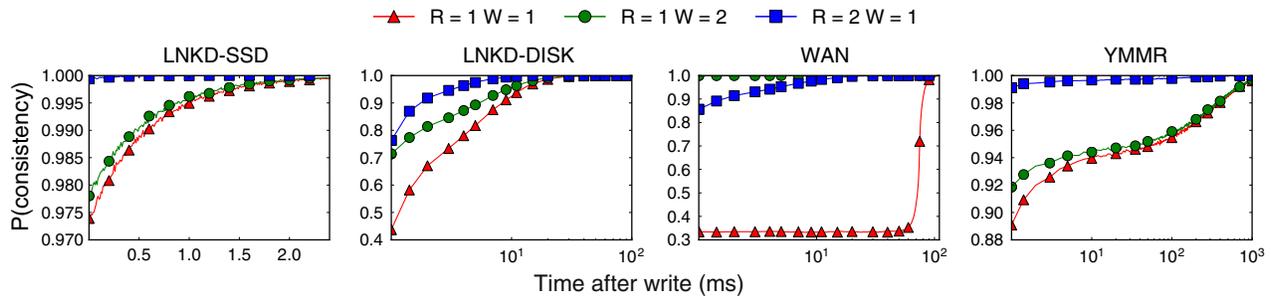


Fig. 6 (Δ, p) -regular semantics for production operation latencies

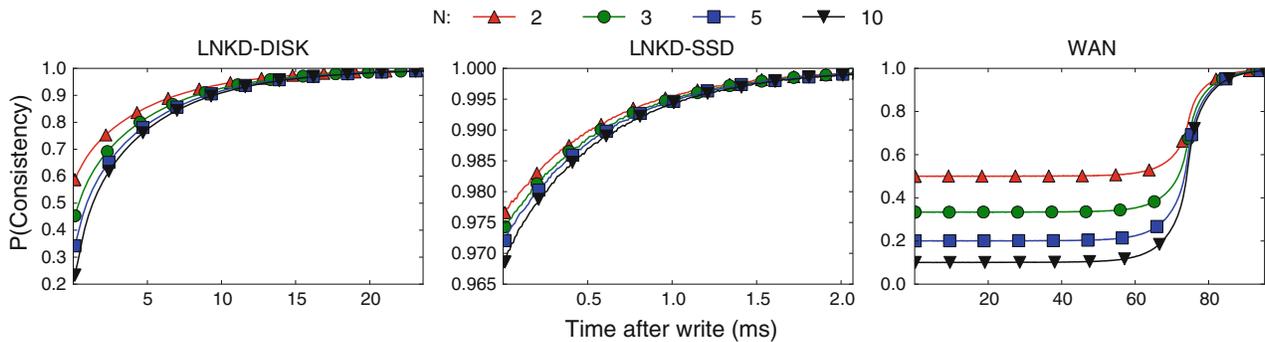


Fig. 7 (Δ, p) -regular semantics for production operating latencies for variable N when $R = W = 1$. Δ for 99.9% probability of consistency is similar across N

6.8 Latency versus staleness

Choosing a value for R and W is a trade-off between operation latency and (Δ, p) -regular semantics. To measure the obtainable latency gains, we compared (Δ, p) -regular semantics required for a 99.9% probability of consistent reads to the 99.9th percentile read and write latencies.

Partial quorums often exhibit favorable latency–consistency trade-offs (Table 4). For YMMR, $R = W = 1$ results

in low latency reads and writes (16.4ms) but high (Δ, p) -regular semantics (1,364ms). However, setting $R = 2$ and $W = 1$ reduces (Δ, p) -regular semantics to 202ms and the combined read and write latencies are 81.1% (186.7ms) lower than the fastest strict quorum ($W = 1, R = 3$). A 99.9% consistent (Δ, p) -regular semantics of 13.6ms reduces LNKD-DISK read and write latencies by 16.5% (2.48ms). For LNKD-SSD, across 10M writes (“seven nines”), we did not observe staleness with $R = 2, W = 1$.

Table 4 (Δ, p) -regular semantics for $p_{st} = .001$ (99.9% probability of consistency for 50,000 reads and writes) and 99.9th percentile read (L_r) and write latencies (L_w) across R and W , $N = 3$ (1 M reads and writes)

	LNKD-SSD			LNKD-DISK			YMMR			WAN		
	L_r	L_w	t	L_r	L_w	t	L_r	L_w	t	L_r	L_w	t
$R = 1, W = 1$	0.66	0.66	1.85	0.66	10.99	45.5	5.58	10.83	1,364.0	3.4	55.12	113.0
$R = 1, W = 2$	0.66	1.63	1.79	0.65	20.97	43.3	5.61	427.12	1,352.0	3.4	167.64	0
$R = 2, W = 1$	1.63	0.65	0	1.63	10.9	13.6	32.6	10.73	202.0	151.3	56.36	30.2
$R = 2, W = 2$	1.62	1.64	0	1.64	20.96	0	33.18	428.11	0	151.31	167.72	0
$R = 3, W = 1$	4.14	0.65	0	4.12	10.89	0	219.27	10.79	0	153.86	55.19	0
$R = 1, W = 3$	0.65	4.09	0	0.65	112.65	0	5.63	1,870.86	0	3.44	241.55	0

Significant latency–staleness trade-offs are in bold

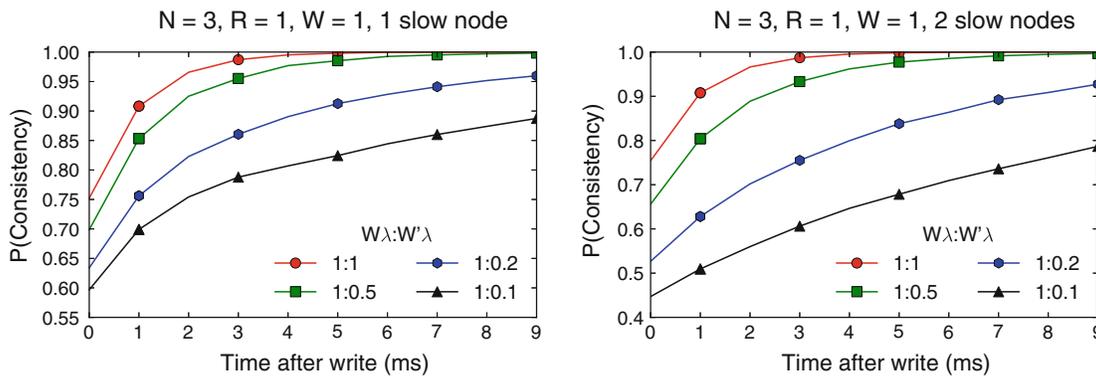


Fig. 8 Impact of heterogeneous synthetic latency distributions across replicas. We evaluate cases where one or two replicas have higher write latency (W') than the other replica(s) (W) for exponentially distributed latencies

$R = W = 1$ reduced latency by 59.5% (1.94ms) with a corresponding (Δ, p) -regular semantics of 1.85 ms. Under WAN, $R > 1$ or $W > 1$ results in a large latency increase because writes incur large wide-area transit delays. In summary, lowering values of R and W can greatly improve operation latency, but, even in the tail, the duration of inconsistency (Δ) is relatively small.

6.9 Heterogeneous replica behavior

In our experiments thus far (with the exception of the multi-datacenter WAN scenario), we have assumed that all replicas behave according to the same latency distribution. In practice, this may not be the case: some nodes may have faulty components, may experience greater network delays, or may otherwise display anomalous behavior. Accordingly, we now use our PBS models to consider how heterogeneous nodes could affect the observed consistency. To begin, we quantify the effects of slow writes at replicas, using a range of exponentially distributed write latencies per replica in our Monte Carlo analysis. We fix $A=R=S$ latencies as exponential distributions with mean 1 ms and vary W . Figure 8 shows how consistency changes with time when we either have one or two slow replicas (for $N = 3, R = 1, W = 1$). With one

slow replica, when the variance of the slower replica (W^*) is 100 ms ($\lambda = .1$, mean 10 ms) the chance of consistent reads immediately after a write drops to around 60% (compared to 75% when all replicas have the same write latency). We see more inconsistency as the variance of the slow replica increases because write variance dictates the amount of staleness observed in the WARS model. Comparing the results for W^* with 100 ms variance to Sect. 6.3 (Fig. 4), we see that having one slow replica (60% chance at $\Delta = 0, \lambda = 0.10$) has lower staleness than all three replicas being slow (40% chance at $\Delta = 0, \lambda = 0.10$). Accordingly, having a few high variance replicas is better than having all high variance replicas.

To explore a real-world setting with heterogeneous replica behavior, we consider a scenario where some of the replicas store data on SSDs while others use spinning disks. We model this setup by using the LNKD-SSD and LNKD-DISK latency distributions for different replica configurations. For $R = 1, W = 1$ (Fig. 9), having one replica with disks and two replicas with SSDs results in (Δ, p) -regular semantics of 35 ms at 99.9% probability of consistency. When two replicas use disks, we find that the (Δ, p) -regular semantics at 99.9% probability becomes 43 ms, which is very close to the case where all the replicas use spinning disks (45.5 ms at 99.9%).

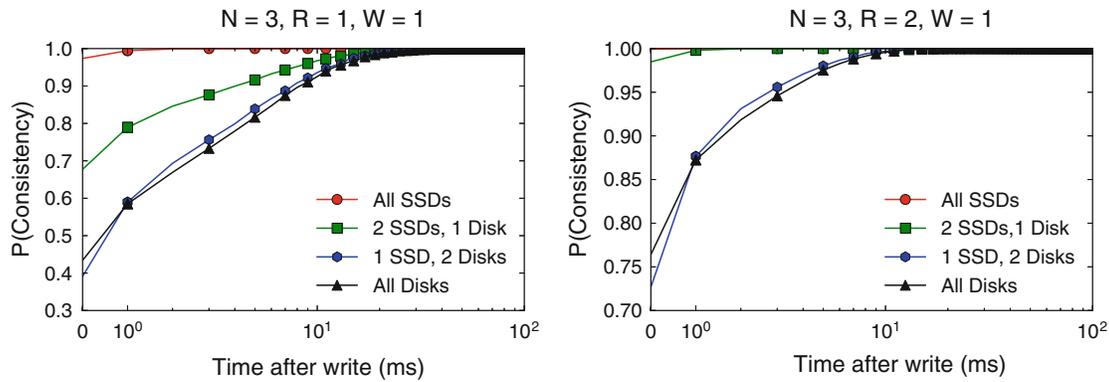


Fig. 9 Impact of heterogeneous empirical latency distributions across replicas. For $N = 3$, we consider cases where some replicas use SSDs while others use disks

With $R = 2, W = 1$, the effect of the slow node is reduced, and the (Δ, p) -regular semantics are 3 ms at 99.9% probability (compared to 13.6 ms at 99.9%). These results indicate that the consistency benefit of partially upgrading hardware across servers may not outweigh the cost of performance heterogeneity. However, if upgrading for other reasons (e.g., latency), consistency may improve as a side benefit.

We also modeled cases where a subset of the replicas have high latencies for W, A, R and S , a plausible approximation of the scenario of network congestion between some server racks within a data center (not shown). When all clients experienced the same slowdown across replicas (i.e., the clients were not co-located with the slow rack), we found that observed consistency actually increased, as the slower replicas were ignored by the read operations. This results in writes and reads being served from the faster replicas, meaning there are fewer chances for message re-ordering. Similarly, a single node failure can be approximated by using a latency distribution with a large mean for all operations on the failed node, and we again see an increase in consistency. If clients and servers are partitioned from one another, we do not expect this behavior: observed consistency will degrade.

7 Multi-key guarantees

In this section, we consider two important kinds of multi-key guarantees: transactional atomicity and causal consistency.

7.1 Transactional atomicity

Transactional atomicity (not to be confused with linearizability, often called atomicity in a distributed context) ensures that either all effects of a transaction are seen, or none are. We can use PBS to provide conservative bounds on the probability of transactional atomicity. If we assume that each write operation in the transaction is independent and all writes are

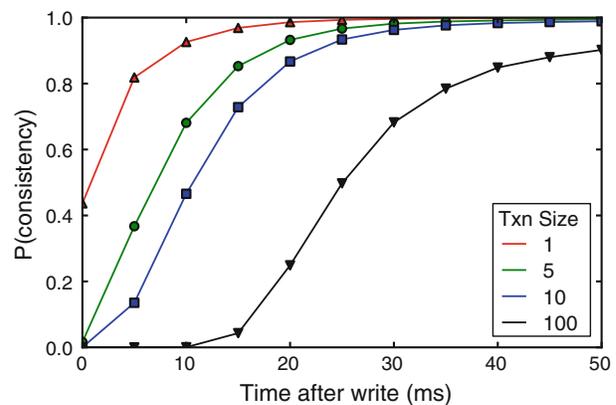


Fig. 10 Probability of transactional atomicity for LNKD-DISK, varying transaction size

buffered until transaction commit, then the probability that we observe transactional atomicity is equal to $P(\text{all updates are visible}) + P(\text{no updates are visible})$.

We consider write-only transactions where we wish to observe all updated data items after commit (such that delaying the visibility of the updated data items is not allowed) and plot the results of varying transaction size in Fig. 10. Under LNKD-DISK, 50 ms after a write, a transaction of size 2 has a 99.4% chance of consistency, while a transaction of size 100 only has a 90.2% chance of consistency. However, after 1 s, a transaction of size 100 has a 99.8% chance of consistency—the LNKD-DISK distribution has a long tail, but it is unlikely that samples drawn from the distribution will take longer than a few seconds.

7.2 Causal consistency

Causal consistency has received considerable attention in recent research on distributed data stores [14, 54]. Under causal consistency, reads obey a partial order of versions

across data items [7]. As an example, a social networking site might wish to enforce that comment replies are seen only along with their parents and would accordingly order replies after their parent comment in the partial order. We can use PBS to determine the likelihood of violating causal consistency in an eventually consistent data store.

To understand the structure of application-level causality relationships, we examined two publicly available datasets from two popular Web services: Twitter and Metafilter. Twitter is a microblogging service with over 500 million users. We use a corpus of 936,236 conversations on Twitter comprising 4,937,001 Tweets collected between February and July 2011 [66]. Metafilter is an active community Weblog with a range of topics and over 59,000 users. We use a corpus of Metafilter posts from July 1999 to October 2012 containing 362,584 posts and 8,749,130 comments [56].

Using PBS predictions, we can determine the likelihood that an eventually consistent store would show events out of order; that is, a Tweet in a conversation might be shown without the rest of the conversation. Effectively, violations occur when an event appears before its respective ancestors in the causal order have arrived. The likelihood of this reordering is determined by the eventually consistent store as well as the inter-arrival time for each event. If we have three events A , B , and C occurring at times t_A , t_B , t_C such that $A \rightarrow B \rightarrow C$ in the desired causal ordering, we can calculate the probability of causal consistency at time $t_D > t_C$ given a measure for (Δ, p) -semantics. As a simple case, consider no overwrites to A , B , or C , and a single set of reads to all three items. If we denote the case in which the read returns a non-null data item I as R_I (and null as $\neg R_I$), then the probability of causal consistency is:

$$\begin{aligned} &P(\neg R_A \wedge \neg R_B \wedge \neg R_C) \\ &+ P(R_A \wedge \neg R_B \wedge \neg R_C) \\ &+ P(R_A \wedge R_B \wedge \neg R_C) \\ &+ P(R_A \wedge R_B \wedge R_C) \end{aligned} \quad (6)$$

We can calculate each of R_A , R_B , and R_C based on t_A , t_B , t_C , and t_D , setting $\Delta = t_D - t_A$, and so on.

In practice, the inter-arrival time between replies is far greater than most Δ for reasonably high p in the (Δ, p) -regular semantics we saw in Sect. 6. We plot the inter-arrival time for each conversation/comment thread in Fig. 11. The median inter-arrival time is 153 s for Twitter and over 412 s for Metafilter. Only 0.001% of Tweet replies arrived within 1 s, and only 0.0766% arrived within 10 s. 0.3693% of Metafilter comments arrived within one second of the prior comment, while 3.1225% arrived within 10 s. When compared to expected stability within hundreds of milliseconds at the 99.9th percentile, these inter-arrival times leave little room for reordering within application-level causal relationships.

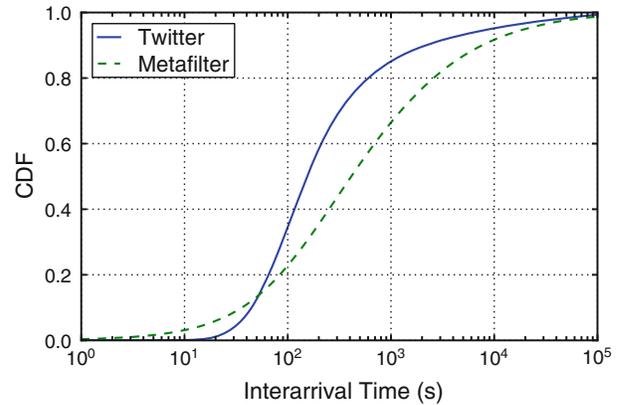


Fig. 11 Inter-arrival time for Twitter and Metafilter causal events

Table 5 Probability of causal consistency for 0.001% fastest inter-arrival times for conversations in Twitter and Metafilter dataset

Model	Twitter		Metafilter	
	0s	1s	0s	1s
LNKD-DISK	95.08 (%)	99.99 (%)	75.35 (%)	99.97 (%)
YMMR	97.94 (%)	99.92 (%)	90.01 (%)	99.64 (%)

Especially as the window of inconsistency closes within hundreds of milliseconds while the majority of requests for a data item often occur much later, we expect causal violations due to eventually consistent store operation to be rare during *normal* operation in practice. We configured our (Δ, p) -regular predictor for LNKD-DISK and YMMR and calculated the probability of causal inconsistency for the 0.001th percentile of Twitter and Metafilter thread prefixes ranked by fastest inter-arrival time (i.e., if $A \rightarrow B \rightarrow C$, B happened immediately after A , then C happened $t > 0$ units of time after B , we only consider $A \rightarrow B$). As shown in Table 5, the probability of causal consistency immediately after the latest event is at least 75% and is at least 99.6% after 1 s for these threads, which are themselves among the fastest in the dataset. Of course, a probabilistic expectation is not a deterministic guarantee, but this data, paired with our PBS results, helps further illuminate how, for many services—particularly those with human-generated causal event chains (which are limited in inter-arrival time by both human processing time and fine motor reactions)—eventually consistent configurations often provide causal consistency.

8 Discussion: PBS design and implementation

8.1 Prediction and verification

There has been a proliferation of recent work verifying and measuring the behavior of various stores with respect to different consistency models [19, 35, 65, 75, 80]. Here, we

briefly discuss the strengths and weaknesses of both consistency prediction and verification.

Prediction, as in PBS, gives an expectation of system behavior given a set of input data about the system and the current operating environment. This prediction is both flexible (allowing a wide range of scenarios and parameter exploration) and decentralized (e.g., PBS prediction can be performed on a single node, or even outside of the system—given input data) but is only as good as the traces captured. With perfectly representative input data, capturing all subtleties like dependencies between requests and across predictions, predictions will be entirely accurate. With unrepresentative data or bad models, predictions may be inaccurate. On the other hand, predictions allow users to easily perform “what-if” analysis across arbitrary replication configurations, request distributions (Δ and K), and hardware configurations (e.g., switching from SSDs to disks). Given a trace, one can determine the staleness after an arbitrary amount of time or number of versions without having to actually run any additional queries, and, in our experience, the computational requirements for prediction are modest (achievable in real-time even in a Web browser—see Sect. 8.3). Algorithmically, prediction does not require consensus: we need not know the “latest version” for any key (Sect. 5.3). This means that readers and writers need not coordinate or all participate in order to provide meaningful statistics for users. In our Cassandra implementation, only one server needs to participate in predictions, even if other servers act as coordinators for requests to the same key.

In contrast with prediction, verification informs users how their data stores are performing with certainty. If a user makes a change to their replication settings using a predictor, she may want to ensure that the change behaves as expected. While this verification is not well suited to the first step (“what-if” analysis) or in determining how the system will behave under different workloads, it is an important complement to prediction. Verification effectively provides a metric that results from integrating the (K, Δ, p) -regular semantics PDF weighted by the given read request rate (measured with respect to time since the last write). If verifying how often an eventually consistent (or weakly consistent) store provides “strong” consistency models such as linearizability or sequential consistency, then, in the presence of partitions, verification will stall. Additionally, verification is algorithmically complex [35] (NP-complete for models such as serializability and linearizability if values written are not unique [33, 71]), but, in our experience (Sect. 6.2) is not terribly difficult to implement.

Taken together, consistency prediction and verification techniques form a powerful toolkit. Prediction allows exploration of arbitrary configurations and both server and end-user behavior, while verification validates predictions. Prediction acts on a set of finite input data and allows reasoning

about arbitrary conditions and traces, while verification—by definition—operates on a finite trace of system behavior. We believe that both techniques will be increasingly useful as systems begin to treat consistency as a continuous, quantitative metric.

8.2 White-box versus black-box techniques

One final considerable distinction between PBS and recent work on consistency verification is the treatment of the underlying data store: should the data store be treated as a transparent white box, or should we disregard any expert knowledge we have in favor of ostensible portability?

In this work, we take a white-box approach to consistency: if we know how data stores are implemented, we can exploit this knowledge. At the minimum, a distributed data store developer has access to its source code and can examine the protocols used in providing predictions. In practice, others (like the authors) can often examine the (open) source code for themselves. Moreover, once the prediction developer has identified the relevant portions of the replication protocol, she can instrument their behavior and expose these metrics in a separate prediction module. Note that, in the white-box model, we translate the user-centric, declarative specification of consistency anomalies into back-end protocol events (e.g., in WARS, the reordering between read and write responses).

In contrast, recent related work treats data stores as black boxes. These verification-centric techniques typically require monitoring end-user requests and reconstructing consistency behavior from a prefix of the operations. The benefit of this approach is that these techniques can work atop arbitrary stores (practical yet substantial issues relating to data model and query interfaces aside). Performing prediction in a black-box scenario appears substantially more complex, possibly requiring inference regarding the behavior of the underlying data store or otherwise constructing a model via active search of the state space. Unlike white-box modeling, which requires a protocol-centric translation of consistency anomalies, black-box modeling is, by definition, constrained to reason about operation traces (or similar), which are closer to the original model definitions.

Given our success in integrating prediction into an existing data store (Sect. 8.3), we believe that white-box techniques are feasible, even if they require modifications to existing stores.

8.3 Real-world store integration

With the help of several open source developers, we have developed patches for PBS functionality within two NoSQL stores: Cassandra and Voldemort. In this section, we briefly outline the technical details and architecture of these modifications. For Cassandra, we have taken two approaches: an

invasive but more accurate implementation and an external but less accurate prediction module. For Voldemort, we have performed simple logging required for reconstructing WARS offline. Next, we discuss the technical details involved in our two approaches to PBS predictions in Cassandra.

Our first approach to predictions in Cassandra modified the messaging layer to directly provide a node-local prediction module with latency traces. We integrated, as part of Cassandra 1.2.0, PBS profiling and prediction. We modified the Cassandra messaging layer to add a message creation timestamp in order to measure each of W , A , R , S distributions. When tracing is enabled on a given node, the messaging layer logs per-operation timestamps in a separate PBS prediction module. The timestamps are stored in an in-memory circular buffer for each of the required message latencies. Subsequently, users can call the PBS predictor module via an externally accessible interface (Java MBean, or via JMX) which they can use to provide advanced functionality like dynamic replication configuration and monitoring (Sect. 8.4). This provides highly accurate predictions at the expense of having to modify the messaging layer. Predictions can be performed outside of the database or as an internal module.

Our second approach is less invasive and leverages several developments that were introduced to the system during the course of this project. Concurrently with our modifications, Cassandra developers introduced improved monitoring for each of its tables (Column Families), including latency sampling. This enables basic (but not necessarily high-fidelity) trace estimation without performing any modifying the underlying message layer (as in our first implementation). Accordingly, stock Cassandra allows rough predictions. However, currently (as of July 2013), Cassandra only allows traces for round-trip times, requiring a predictor to estimate the time spent in the network and in local processing. Given the benefits of fine-grained per-replica monitoring (including ascertaining how query performance is affected by the network and the storage manager), we believe that the remaining metrics required for accurate predictions will be both easy to implement and useful to other downstream consumers (i.e., traditional monitoring tools). Accordingly, while our second implementation, a *external* predictor that consumes standard Cassandra metrics, is currently limited to round-trip times, we expect this limitation to be lifted in the near future. As an alternative design, we have considered leveraging Cassandra's experimental query tracing features to provide more accurate timing.

If Cassandra is a case study for PBS prediction, we are encouraged by the rapid pace of feature development over the past 18 months. Not only is the feature set of these emerging data stores growing, but, in our experience, developers are willing to engage with academics throughout the design and implementation phases. Ultimately, balancing the challenge of code maintainability with experimental functionality is

an engineering exercise, but we believe that, as these stores mature, our ability to make white-box predictions will only improve.

8.4 Additional functionality

8.4.1 Latency-consistency SLAs

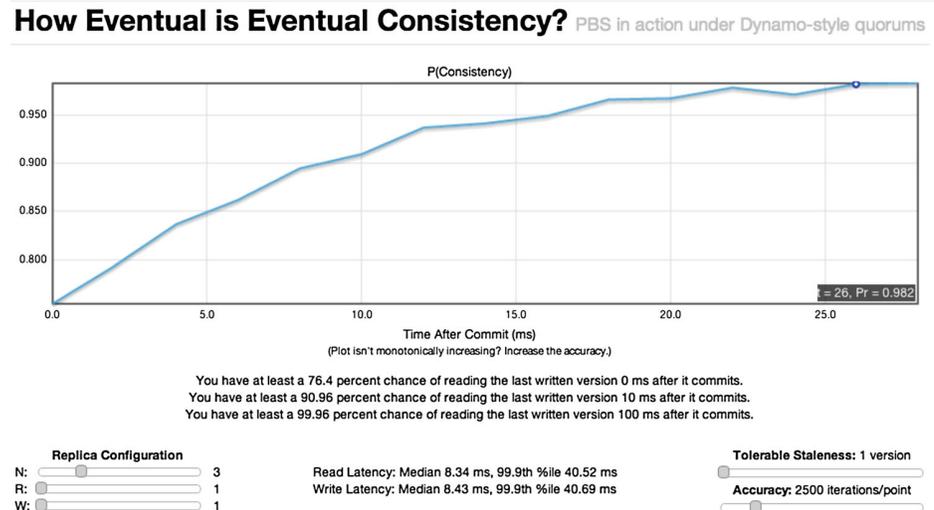
With PBS, we can automatically configure replication parameters by optimizing operation latency given constraints on staleness and minimum durability. Data store operators can subsequently provide service level agreements to applications and quantitatively describe latency/staleness trade-offs to users. Operators can dynamically configure replication using online latency measurements. PBS provides a quantitative lens for analyzing consistency guarantees that were previously unknown. This optimization formulation is likely non-convex, but the state space for configurations is small ($O(N^2)$). This optimization also allows disentanglement of replication for reasons of durability from replication for reasons of low latency and higher capacity. For example, operators can specify a minimum replication factor for durability and availability but can also automatically increase N , decreasing tail latency for fixed R and W .

To illustrate these capabilities, we developed a demonstration of these functionalities for developers, which we posted online as an interactive browser-based console (Fig. 12). The demonstration shows how a hypothetical user or systems administrator can—today, with the aid of PBS predictions in her database—explore the state space of configurations without changing the database's settings in production. While the demo requires a human to operate, as we have alluded to, we believe that automating the search process and configuration selection can be easily automated in the future; we briefly outlined these possibilities in a SIGMOD 2013 demo that featured similar real-time predictions, but for a live Cassandra cluster and Web service [16].

8.4.2 On-boarding

Our discussions with industry indicate that PBS could also be used as an aid for on-boarding internal customers of an eventually consistent data store. When developers wish to leverage an internally deployed data store, they frequently work with data store administrators and support staff on provisioning and configuration. When faced with decisions about replication parameters, developers often do not have insight into how their data store will perform or what semantics the store will provide in an eventually consistent deployment. With PBS, this on-boarding process is simplified and the developer—who likely has a rough cost-benefit model for latency-consistency—can make an informed decision (à la consistency rationing [47]).

Fig. 12 Screenshot of PBS configuration demo (Hosted online at <http://pbs.cs.berkeley.edu/>)



9 Related work

We surveyed quorum replication techniques [4–6, 8, 9, 34, 36, 41, 44, 58, 60, 61] in Sect. 3. In this work, we specifically draw inspiration from probabilistic quorums [58] and deterministic k -quorums [8, 9] in analyzing expanding quorum systems and their consistency. We believe that revisiting probabilistic quorum systems—including non-majority quorum systems such as tree quorums—in the context of write propagation, anti-entropy, and Dynamo is a promising area for theoretical work.

Maintaining data consistency is a long-studied problem in distributed systems [25] and concurrent programming [42]. There is a plethora of consistency models offering different trade-offs between semantics, performance, and availability. Traditional models like serializability [64] and linearizability [42] as well as more recently proposed models such as timeline consistency [24] and parallel snapshot isolation [68] all provide “strong” semantics at the cost of high availability, or the ability to provide “always-on” response behavior at all replicas. In contrast, faced with a requirement for high availability and low latency, many production data stores have turned to weaker semantics to provide availability in the face of partitions [25, 74].

Our focus in this paper is on the semantics provided by existing, widely deployed systems both in theory and in practice. Due to the prevalence of “strong” consistency and eventual consistency models in practice (and the explicit choice between these two models in Dynamo-style systems), we largely focus on this dichotomy. However, there are a range of alternative but still “weak” models. As an example, the Bayou system provided a range of “session guarantees,” including read-your-writes and monotonic reads consistency [72]. Similarly, a technical report from UT Austin claims that a variant of causal consistency is the strongest consistency model

achievable in an available, one-way convergent (eventually consistent) system [57], a model that has recently attracted systems implementations [54]. As we have hinted, probabilistic approaches are applicable to the consistency models beyond those we have considered here.

Prior research has examined how to provide deterministic staleness bounds. FRACS [81] allows replicas to buffer updates up to a given staleness threshold under multiple replication schemes, including master-slave and group gossip. AQUA [48] asynchronously propagates updates from a designated master to replicas that in turn serve reads with bounded staleness. AQUA actively selects which replicas to contact depending on response time predictions and a guaranteed staleness bound. TRAPP [62] provides trade-offs between precision and performance for continuously evolving numerical data. TACT [78, 79] models consistency along three axes: numerical error, order error, and staleness. TACT bounds staleness by ensuring that each replica (transitively) contacts all other replicas in the system within a given time window. Finally, PIQL [12] bounds the number of operations performed per query, trading operation latency at scale with the amount of data a particular query can access, impacting accuracy. These deterministically bounded staleness systems represent the deterministic dual of PBS.

Finally, recent research has focused on measuring and verifying the consistency of eventually consistent systems both theoretically [35] and experimentally [19, 65, 75, 80]. This is useful for validating consistency predictions and understanding staleness violations.

10 Conclusion

In this paper, we introduced Probabilistically Bounded Staleness, which models the expected staleness of data returned by

eventually consistent data stores. PBS offers an alternative to the all-or-nothing consistency guarantees of today's systems by offering SLA-style consistency predictions. By extending prior theory on probabilistic quorum systems, we derived an analytical solution for the (K, p) -regular semantics of a partial quorum system, representing the expected staleness of a read operation in terms of versions. We also analyzed (Δ, p) -regular semantics, or expected staleness of a read in terms of real-time, under Dynamo-style quorum replication. To do so, we developed the *WARS* latency model to explain how message reordering leads to staleness under Dynamo. To examine the effect of latency on (Δ, p) -regular semantics in practice, we used real-world traces from Internet companies to drive a Monte Carlo analysis. We find that eventually consistent quorum configurations are often consistent after tens of milliseconds due in large part to the resilience of Dynamo-style protocols. We conclude that “eventually consistent” partial quorum replication schemes frequently deliver consistent data while offering significant latency benefits.

11 Interactive demonstration

An interactive demonstration of Dynamo-style PBS is available at <http://pbs.cs.berkeley.edu/#demo>.

Acknowledgments The authors would like to thank Alex Feinberg and Coda Hale for their cooperation in providing real-world distributions for experiments and for exemplifying positive industrial–academic relations through their conduct and feedback. The authors would also like to thank the following individuals whose discussions and feedback improved this work: Marcos Aguilera, Peter Alvaro, Eric Brewer, Neil Conway, Aaron Davidson, Greg Durrett, Jonathan Ellis, Andy Gross, Hariyadi Gunawi, Sam Madden, Bill Marczak, Kay Ousterhout, Vern Paxson, Mark Phillips, Christopher Ré, Aviad Rubenstein, Justin Sheehy, Scott Shenker, Sriram Srinivasan, Doug Terry, Anirudh Todi, Greg Valiant, and Patrick Wendell. We would especially like to thank Bryan Kate for his extensive comments and Ali Ghodsi, who, in addition to providing feedback, originally piqued our interest in theoretical quorum systems. This work was supported by gifts from Google, SAP, Amazon Web Services, Blue Goji, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, IBM, Intel, MarkLogic, Microsoft, NEC Labs, NetApp, NTT Multimedia Communications Laboratories, Oracle, Quanta, Splunk, and VMware. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant DGE 1106400, National Science Foundation Grants IIS-0713661, CNS-0722077 and IIS-0803690, NSF CISE Expeditions award CCF-1139158, the Air Force Office of Scientific Research Grant FA95500810352, and by DARPA contract FA865011C7136.

References

1. Apache Cassandra 1.0 documentation: About Data Consistency in Cassandra. http://datastax.com/docs/1.0/dml/data_consistency
2. Apache Cassandra Jira: cassandra-876: Support Session (Read-After-Write) Consistency. <https://issues.apache.org/jira/browse/CASSANDRA-876>. October 2010. Accessed 13 Dec 2011

3. Abadi, D.J.: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Comput.* **45**(2), 37–42 (2012)
4. Abraham, I., Malkhi, D.: Probabilistic quorums for dynamic systems (extended abstract). In: *DISC*, pp. 60–74 (2003)
5. Agrawal, D., Abbadi, A.E.: The tree quorum protocol: An efficient approach for managing replicated data. In: *VLDB*, pp. 243–254 (1990)
6. Agrawal, D., Abbadi, A.E.: Resilient logical structures for efficient management of replicated data. In: *VLDB*, pp. 151–162 (1992)
7. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.: Causal memory: definitions, implementation and programming. *Distrib. Comput.* **9**(1), 37–49 (1995)
8. Aiyer, A., Alvisi, L., Bazzi, R.A.: On the availability of non-strict quorum systems. In: *DISC*, pp. 48–62 (2005)
9. Aiyer, A.S., Alvisi, L., Bazzi, R.A.: Byzantine and multi-writer k-quorums. In: *DISC*, pp. 443–458 (2006)
10. Alpern, B., Schneider, F.B.: Defining liveness. *Inf. Process. Lett.* **21**(4), 181–185 (1985)
11. Alvaro, P., Conway, N., Hellerstein, J.M., Marczak, W.R.: Consistency analysis in Bloom: a CALM and collected approach. In: *CIDR*, pp. 249–260 (2011)
12. Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M.J., Patterson, D.A.: PIQL: Success-tolerant query processing in the cloud. In: *VLDB*, pp. 181–192 (2012)
13. Basho Riak: <http://basho.com/products/riak-overview/> (2012)
14. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: The potential dangers of causal consistency and an explicit solution. In: *SOCC* (2012)
15. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. *PVLDB* **5**(8), 776–787 (2012)
16. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Pbs at work: advancing data management with consistency metrics. In: *SIGMOD* (2013). Demo
17. Basho Technologies, Inc.: Riak wiki: Riak concepts replication. <http://wiki.basho.com/Replication.html>. Accessed Jan 2013
18. Basho Technologies, Inc.: riak_kv 1.0 application. https://github.com/basho/riak_kv/blob/1.0/src/riak_kv_app.erl
19. Bermbach, D., Tai, S.: Eventual consistency: how soon is eventual? An evaluation of Amazon S3's consistency behavior. In: *MW4SOC*, pp. 1:1–1:6 (2011)
20. Birman, K., Chockler, G., van Renesse, R.: Toward a cloud computing research agenda. *SIGACT News* **40**(2), 68–80 (2009)
21. Blomstedt, J.: Bringing consistency to Riak. Talk at RICON 2012 (<http://vimeo.com/51973001>)
22. Cassandra 1.0 Thrift Configuration. <https://github.com/apache/cassandra/blob/cassandra-1.0/interface/cassandra.thrift>
23. Cassandra wiki: Operations. http://wiki.apache.org/cassandra/Operations#Repairing_missing_or_inconsistent_data. Accessed Jan 2013
24. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* **1**(2), 1277–1288 (2008). <http://dl.acm.org/citation.cfm?id=1454159.1454167>
25. Davidson, S., Garcia-Molina, H., Skeen, D.: Consistency in partitioned networks. *ACM Comput. Surv.* **17**(3), 314–370 (1985)
26. Dean, J.: Designs, lessons, and advice from building large distributed systems. In: Keynote from LADIS (2009)
27. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: *SOSP*, pp. 205–220 (2007)

28. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: PODC, pp. 1–12 (1987)
29. Ellis, J.B.: Revision 986783: revert 'per-connection read-your-writes "session" consistency'. <http://svn.apache.org/viewvc?view=revision&revision=986783>. 18 August 2010, one week after the original patch was accepted
30. Feinberg, A.: Personal communication. 23, 24 October, 14, 19, 21, 30 November, 1 December 2011
31. Feinberg, A.: Project Voldemort: Reliable distributed storage. In: ICDE (2011). Project site: <http://www.project-voldemort.com> (2012)
32. Fu, A.W.: Delay-optimal quorum consensus for distributed systems. *IEEE Trans. Parallel Distrib. Syst.* **8**(1), 59–69 (1997)
33. Gibbons, P.B., Korach, E.: Testing shared memories. *SIAM J. Comput.* **26**(4), 1208–1244 (1997)
34. Gifford, D.K.: Weighted voting for replicated data. In: SOSP, pp. 150–162 (1979)
35. Golab, W., Li, X., Shah, M.A.: Analyzing consistency properties for fun and profit. In: PODC, pp. 197–206 (2011)
36. Gupta, A., Maggs, B.M., Oprea, F., Reiter, M.K.: Quorum placement in networks to minimize access delays. In: PODC, pp. 87–96 (2005)
37. Hale, C.: Personal Communication. 16 November 2011
38. Hale, C., Kennedy, R.: Using Riak at Yammer. http://dl.dropbox.com/u/2744222/2011-03-22_Riak-At-Yammer.pdf. 23 March 2011
39. Hamilton, J.: Perspectives: I love eventual consistency but...<http://perspectives.mvdirona.com/2010/02/24/ILoveEventualConsistencyBut.aspx>. 24 February 2010
40. Helland, P., Campbell, D.: Building on quicksand. In: CIDR (2009)
41. Herlihy, M.: Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.* **12**(2), 170–194 (1987)
42. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990)
43. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX ATC, pp. 145–158 (2010)
44. Jiménez-Peris, R., Patiño Martínez, M.: Are quorums an alternative for data replication? *ACM Trans. Database Syst.* **28**(3), 257–294 (2003)
45. King, D.: keltranis comment on "reddit's now running on Cassandra". http://www.reddit.com/r/programming/comments/bcqh/reddits_now_running_on_cassandra/c0m3wh6. March 2010
46. Kirkell, J.: Consistency or bust: breaking a Riak cluster. <http://www.oscon.com/oscon2011/public/schedule/detail/19762>. Talk at O'Reilly OSCON 2011, 27 July 2011
47. Kraska, T., Hentschel, M., Alonso, G., Kossmann, D.: Consistency rationing in the cloud: pay only when it matters. In: Proceedings of the VLDB Endowment, vol. 2, issue 1, pp. 253–264 (2009)
48. Krishnamurthy, S., Sanders, W.H., Cukier, M.: An adaptive quality of service aware middleware for replicated services. *IEEE Trans. Parallel Distrib. Syst.* **14**(11), 1112–1125 (2003)
49. Lakshman, A., Malik, P.: Cassandra—a decentralized structured storage system. In: LADIS, pp. 35–40 (2008). Project site: <http://cassandra.apache.org> (2012)
50. Lamport, L.: On interprocess communication. *Distrib. Comput.* **1**(2), 86–101 (1986)
51. Lamport, L.: The part-time parliament. *ACM Trans. Comput. Syst.* **16**(2), 133–169 (1998)
52. Linden, G.: Make Data Useful. <https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-29.ppt>. 29 November 2006
53. Linden, G.: Marissa Mayer at Web 2.0. <http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html>. 9 November 2006
54. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: SOSP, pp. 401–416 (2011)
55. Lynch, J.: Rolling with Riak. <http://sdruby.org/podcast/81>. Talk presented at SD Ruby meeting (Podcast 81), 2010
56. Metafilter Infodump: <http://stuff.metafilter.com/infodump/>. Combination of all available comment datasets: mefi, askme, meta, music. User count from usernames
57. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, convergence. Tech. Rep. TR-11-22, Computer Science Department, University of Texas at Austin (2011)
58. Malkhi, D., Reiter, M., Wool, A., Wright, R.: Probabilistic quorum systems. *Inf. Comput.* **170**(2), 184–206 (2001)
59. Marcus, A.: The NoSQL ecosystem. In: The Architecture of Open Source Applications, pp. 185–205 (2011)
60. Merideth, M., Reiter, M.: Selected results from the latest decade of quorum systems research. In: Replication, LNCS, vol. 5959, pp. 185–206. Springer, Berlin (2010)
61. Naor, M., Wool, A.: The load, capacity, and availability of quorum systems. *SIAM J. Comput.* **27**(2), 214–225 (1998)
62. Olston, C., Widom, J.: Offering a precision-performance tradeoff for aggregation queries over replicated data. In: VLDB, pp. 144–155 (2000)
63. Outbrain Inc.: Introduction to no:sql [sic] and Cassandra (and Outbrain). https://docs.google.com/present/view?id=ahbp3bktzpkc_220f7v26vg7. January 2010
64. Papadimitriou, C.: The serializability of concurrent database updates. *J. ACM (JACM)* **26**(4), 631–653 (1979)
65. Rahman, M., Golab, W., AuYoung, A., Keeton, K., Wylie, J.: Toward a principled framework for benchmarking consistency. In: Proceedings of the 8th Workshop on Hot Topics in System Dependability (2012)
66. Ritter, A., Cherry, C., Dolan, B.: Unsupervised modeling of Twitter conversations. In: HLT (2010)
67. Schurman, E., Brutlag, J.: Performance related changes and their user impact. Presented at Velocity Web Performance and Operations Conference (June 2009)
68. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP, pp. 385–400 (2011)
69. Stonebraker, M.: Urban Myths About SQL. http://voldb.com/_pdf/VoltDB-MikeStonebraker-SQLMythsWebinar-060310.pdf. VoltDB Webinar (June 2010)
70. Sumbaly, R.: Writing Own Client for Voldemort. <https://github.com/voldemort/voldemort/wiki/Writing-own-client-for-Voldemort>. 16 June 2011. Accessed 21 Dec 2011
71. Taylor, R.N.: Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica* **19**, 57–84 (1983)
72. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: PDIS, pp. 140–149 (1994)
73. Torres-Rojas, F.J., Ahamad, M., Raynal, M.: Timed consistency for shared distributed objects. In: PODC 1999, pp. 163–172
74. Vogels, W.: Eventually consistent. *CACM* **52**, 40–44 (2009)
75. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storage: the consumers perspective. In: CIDR, pp. 134–143 (2011)
76. Wester, B., Cowling, J., Nightingale, E.B., Chen, P.M., Flinn, J., Liskov, B.: Tolerating latency in replicated state machines through client speculation. In: NSDI, pp. 245–260 (2009)
77. Williams, D.: HBase vs Cassandra: Why We Moved. <http://ria101.wordpress.com/2010/02/24/hbase-vs-cassandra-why-we-moved>. 24 February 2010

78. Yu, H., Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.* **20**(3), 239–282 (2002)
79. Yu, H., Vahdat, A.: The costs and limits of availability for replicated services. *ACM Trans. Comput. Syst.* **24**(1), 70–113 (2006)
80. Zellag, K., Kemme, B.: How consistent is your cloud application? In: *SOCC* (2012)
81. Zhang, C., Zhang, Z.: Trading replication consistency for performance and availability: an adaptive approach. In: *ICDCS*, pp. 687–695 (2003)