Client-centric Benchmarking of Eventual Consistency for Cloud Storage Systems

Wojciech Golab*, Muntasir Raihan Rahman†, Alvin AuYoung‡, Kimberly Keeton‡, Indranil Gupta†
*University of Waterloo, †University of Illinois at Urbana-Champaign, ‡HP Labs

***Abstract*—Eventually-consistent key-value storage systems sacrifice the ACID semantics of conventional databases to achieve superior latency and availability. However, this means that client applications, and hence end-users, can be exposed to stale data. The degree of staleness observed depends on various tuning knobs set by application developers (customers of key-value stores) and system administrators (providers of key-value stores). Both parties must be cognizant of how these tuning knobs affect the consistency observed by client applications in the interest of both providing the best end-user experience and maximizing revenues for storage providers. Quantifying consistency in a meaningful way is a critical step toward both understanding what clients actually observe, and supporting consistency-aware service level agreements (SLAs) in next generation storage systems.**

This paper proposes a novel consistency metric called Γ (Gamma) that captures client-observed consistency. This metric provides quantitative answers to questions regarding observed consistency anomalies, such as how often they occur and how bad they are when they do occur. We argue that Γ is more useful and accurate than existing metrics. We also apply Γ to benchmark the popular Cassandra key-value store. Our experiments demonstrate that Γ is sensitive to both the workload and client-level tuning knobs, and is preferable to existing techniques which focus on worst-case behavior.

I. Introduction

Many cloud products and services such as Web search, e-commerce, and social networking, have to deal with big data. In order to scale with growing amounts of data and numbers of users, the design of these systems has moved away from using conventional ACID databases, toward a new generation of scalable storage systems called *key-value stores* (often categorized more broadly as NoSQL storage systems). Many of these key-value storage systems offer a weak notion of consistency called *eventual consistency* [1], [2]. This stems from Brewer's CAP principle [3], which dictates that a storage system choose between either (strong) consistency or availability during failures that partition the network connecting the storage nodes. Even when the network is reliable, weak consistency is often chosen voluntarily to reduce the latency of storage operations [4].

However, parting with strong consistency exposes client applications to consistency anomalies such as stale reads, which can frustrate end-users. Making client applications robust against such anomalies is difficult in general, but in many cases the anomalies are tolerable, especially if they are minor and infrequent. Thus, balancing latency with consistency is critical to producing a satisfying end-user experience. To strike the right balance, application developers and system administrators need appropriate tools to determine what effect various configuration parameters and tuning knobs have on the consistency observed by client applications. This requires a way to quantify consistency anomalies.

Several techniques have been proposed in the literature for measuring eventual consistency. These suffer from two shortcomings. First, many of them are *system-centric*, i.e., they measure consistency anomalies that reflect the internal state of the storage system more than what a client application might observe. Specifically, several known techniques (e.g., [5], [6], [7]) measure the time between the write of a value and the synchronization of the write among its replicas. However, as we demonstrate in this paper, this system-centric approach paints quite a pessimistic picture of what a client application might observe.

Second, existing techniques do not account for the error inherent in measuring consistency from several vantage points, such as a collection of storage servers and clients. In practice, simple get and put (read and write) operations are not instantaneous – instead they execute over *intervals* of time whose length depends on configuration parameters such as the number of replicas accessed, and situational factors such as network latency or server load. Thus, time differences between pairs of storage operations are inherently difficult to quantify. Clock skew across machines makes this problem worse.

This paper proposes a new *client-centric* metric for benchmarking consistency. Our metric, called Γ (Gamma), can be used to quantify both the frequency of client-observed consistency anomalies, as well as their severity. This, in turn, helps answer questions such as "how stale are the data received by client reads"? Thus, our approach enables empirical evaluation of eventual consistency in arbitrary systems, under arbitrary workloads. We compare and contrast Γ against system-centric consistency metrics.

Concretely, the technical contributions of this paper are:

- We define a novel deterministic client-centric consistency metric called Γ (Gamma). Our metric is inspired by the Δ (Delta) metric [8], [9]. The advantage of Γ over Δ is two-fold: it yields less noisy values, and it makes fewer assumptions than Δ. The latter point, which we explain in Section III, makes Γ more robust against clock skew. Furthermore, Γ enables more comprehensive benchmarking of eventual consistency than existing work on metrics [10], [11], [12], [13] and measurements [5], [6], [7].
- We apply Γ to perform an offline analysis of Cassandra, a popular open-source key-value store [14]. Using Γ,

This work was supported by the NSERC Discovery Grants Program, Hewlett-Packard Labs, AFOSR/AFRL grant FA8750-11-2-0084, and NSF grant CCF 0964471.

we demonstrate how parameters of the configuration and workload affect the actual consistency observed by storage clients. In particular, using Γ, we quantify the impact on observed consistency due to (i) critical Cassandra parameters like the replication factor and client-side read/write consistency levels, and (ii) client workload parameters like key distribution, read/write ratio and throughput (operations/s).

- We demonstrate that a client-centric metric yields more meaningful results than a system-centric metric. Specifically, our experiments reveal that the actual consistency faced by clients can be very different from what system-centric metrics predict. For instance, while replicas may take hundreds of milliseconds to synchronize, in some of our experiments we did not observe any consistency anomalies beyond the margin of error due to clock skew ($\approx$10ms).

II. Background and Related work

The term *consistency* refers to the notion that different clients accessing a storage system agree in some way on the state of data, for example the latest state or the set of transitions giving rise to that state. Ensuring consistency is inherently difficult for two reasons: First, storage systems tend to maintain multiple replicas of data for fault tolerance and performance. Second, storage operations may involve multiple data items or objects. As a result, many systems provide weak consistency, which is difficult to reason about. A number of techniques have been proposed in different contexts for specifying, verifying, measuring and predicting weak consistency. In this section we give an overview of such techniques, drawing on prior work in distributed computing theory, storage systems, and databases.

a) Specification: Distributed computing theory offers a number of concepts to describe the consistency of shared memory. Lamport proposed the notions of *safe*, *regular* and *atomic registers*—shared objects that support read and write operations [15]. The term *atomic* or *atomicity* in this context means that operations applied to a register can be totally ordered so that: (1) the total order extends the "happens before" partial order (i.e., if operation A ends before operation B begins, then A precedes B in the total order); and (2) each read returns the value assigned by the latest write in the total order. Herlihy and Wing later generalized this property to arbitrary object types by defining the *linearizability* property [16].

Recent research has considered weaker forms of atomicity that parameterize the degree of re-ordering among operations. Aiyer et al. defined k-atomicity, which generalizes Lamport's atomicity by allowing a bounded degree of *version-based staleness*: each read may return the value assigned by one of the k most recent writes [10]. In contrast, the Δ-*atomicity* property allows a bounded degree of *time-based staleness* [8]. More precisely, Δ-*atomicity* allows a read to return a value that was considered current up to Δ time units before the read began. Lamport's atomicity property is equivalent to k-atomicity with $k = 1$, and to Δ-atomicity with $\Delta = 0$.

In databases, *transaction isolation* properties are the moral equivalent of the consistency properties discussed herein. The most widely studied isolation level is *serializability* [17], which requires that transactions appear to execute in some serial order but (in principle) allows queries to return stale data [17]. *Strict serializability* [18] further requires that this serial order extend the "happens before" partial order, similarly to linearizability. Google's Spanner database provides *external consistency*, which is similar to strict serializability [19]. Pileus supports a spectrum of consistency properties specified using consistency-based service level agreements (SLAs) [20].

Many storage systems provide *eventual consistency* (e.g., [1], [14], [21]), which guarantees that "if no new writes are made to the object, eventually all accesses will return the last written value" [2]. This property is weak enough to be attainable in highly-available partition-tolerant systems that (according to Brewer's "CAP" principle) preclude strong consistency properties such as linearizability [3], [22]. Formal definitions of eventual consistency for general systems appear in [12], [13]. Shapiro et al. present a formal treatment of eventual consistency in the case when operations on shared objects are commutative and hence avoid "conflicts" by design [23], [24]. Conventional key-value storage systems (e.g., [14], [21]) fall outside this category because they support write operations, which are inherently conflict-prone. That is, the latest value of an object depends on the order of writes applied.

Yu and Vahdat propose TACT (Tunable Availability and Consistency Tradeoffs), a middleware layer that supports application-dependent consistency semantics expressed as a vector of metrics defined over a logical consistency unit or *conit* [25]. Inconsistencies in the observed value of a conit are bounded using three metrics: numerical error, order error, and staleness. For example, a conit can represent messages posted in a social networking site, and its numerical value can be the number of messages posted. In that case, numerical error and order error bound the difference in the number and order of messages received by a given replica versus messages applied globally, and staleness bounds the age of the oldest message that has been applied globally but not seen by a given replica. The Δ metric of Golab, Li and Shah [8] can be regarded as a formal interpretation of Yu and Vahdat's staleness metric in the special case when a conit is a single key-value pair.

b) Verification: Given a specification of a consistency property, it is natural to ask whether a system provides this property. This question can be cast as a formal decision problem whose input is an *execution trace* of a system that records the start and finish times (e.g., measured at clients), the arguments and the response of every operation applied to the system. Gibbons and Korach showed that this problem is NP-complete for sequential consistency and linearizability [26]. However, in the special case when any two write operations on the same object always assign distinct values, linearizability and Δ-atomicity can be verified in polynomial time [26], [8].

c) Measurement: In the traditional sense, a consistency property (e.g., linearizability) is something that a system either provides or fails to provide. Thus, the property can be verified, but not measured. However, eventual consistency is so loosely specified that several attempts have been made to quantify how far it deviates from the gold standard of atomicity. This question can be posed with respect to *client-centric* consistency, which captures what client applications observe directly, or *system-centric* consistency, which captures the convergence of the storage system's replication protocol.

Measuring eventual consistency usually entails quantifying the staleness of data returned by reads, relative to the latest or freshest data. Wada et al. [7] measure time-based staleness in key-value storage platforms by *actively* writing timestamps to a key from one client, and reading the same key repeatedly from another. The time difference between the reader's local time and the writer's timestamp answers the question "how eventual?" Bermbach et al. [6] and Patil et al. [5] apply similar techniques. This general methodology stress-tests the replication protocol of the storage system by applying reads back-to-back with writes at different replicas, and for that reason we consider it system-centric.

A *passive* methodology for benchmarking eventual consistency has also been proposed, in which staleness is computed off-line from an execution trace recorded by clients [9]. Rather than stress-testing the storage system, this approach aims to quantify the actual staleness observed by clients under natural conditions. Thus, the technique is client-centric. Given an arbitrary execution trace, staleness is calculated by finding the minimum number Δ such that the trace is Δ-atomic [8]. The computation entails evaluating a *score function*, which can be used to visualize fluctuations in staleness over time. We also use a client-centric passive methodology in this paper, but with a novel consistency metric Γ that improves upon Δ in environments with loosely synchronized clocks. Furthermore, we provide an extensive sensitivity analysis of Γ to a variety of configuration and workload parameters, whereas only a single trace is analyzed using the Δ metric in [9].

Calculations based upon an execution trace can also be used to quantify the number of consistency anomalies observed. Golab, Li and Shah [8] use dynamic programming to count the number of values involved in such anomalies. Zellag and Kemme [27] instead analyze cycles in *dependency graphs*, similar to graphs that are used to characterize serializability.

d) Prediction: Bailis et al. [11] provide a probabilistically bounded staleness (PBS) model, which predicts staleness using an abstract model of the storage system. This model is parameterized with details such as the distribution of latencies at various processing stages, and used to estimate a function that maps the read-after-write time to the probability of the read returning the latest value. The problem of deducing the read-after-write time, for example from the workload, is left open. PBS focuses on the behavior of failure-free systems, but node failures are considered in follow-up work [28].

III. Consistency Metric

In this section we define a novel metric called Γ (Gamma) for measuring eventual consistency in key-value storage systems. Our metric and the underlying methodology build on the Δ metric [8], [9]. Γ is theoretically sound and can be used to quantify both the frequency and severity of consistency anomalies. We first review the known metric Δ and point out its disadvantages, and then describe our new metric Γ.

A. Prior work: Δ metric

The Δ metric, expressed in units of time, captures the "deviation" of a given execution trace from Lamport's atomicity property. Before we formally define Δ, we introduce some terminology and definitions. For any key k and value x, the operation k.put(x) denotes a write of x to k, and operation k.get():x denotes a read of x from k. For any operation op, the *start time* $op.s$ refers to the time point when the request was initiated by the client, and the *end time* $op.e$ refers to when the request completes at the client. We say that operation op *happens before* operation op', written as $op < op'$, if $op.e < op'.s$. If neither $op < op'$ nor $op' < op$, we say that op and op' are *concurrent*. The "happens before" partial order is used to define Lamport's atomicity property with respect to an execution trace, as described in Section II.

Given an execution trace T, we say that T is Δ-*atomic* if it can be made atomic by *shifting* the start time (i.e., $op.s$) of each get operation to the left by Δ time units. The process of shifting operations is a mathematical abstraction and is performed in off-line analysis; it does not affect the storage system or workload from which the trace was obtained. Also note that according to the definition of Δ, only get operations are considered for shifting, and only at the left endpoint—a point we address in more detail when we contrast Δ against our novel metric Γ.

An example of Δ atomicity is shown in Figure 1 (a), which considers a trace comprising three operations on a single key k, possibly applied at three different clients. As regards atomicity, the "happens before" order in this trace dictates that k.put(1) takes effect before k.put(2), which takes effect before the get operation. Thus, the latest value with respect to the get is 2, and so the return value of 1 indicates an atomicity violation. However, the trace is Δ-atomic for Δ equal to (or greater than) the gap Y between the k.put(2) and k.get():1. This is because shifting the start time of the get to the left by at least Y time units yields an atomic trace, as shown in Figure 1 (b), in which the "happens before" order no longer constrains k.put(2) relative to k.get(). More details and examples are described in [9].

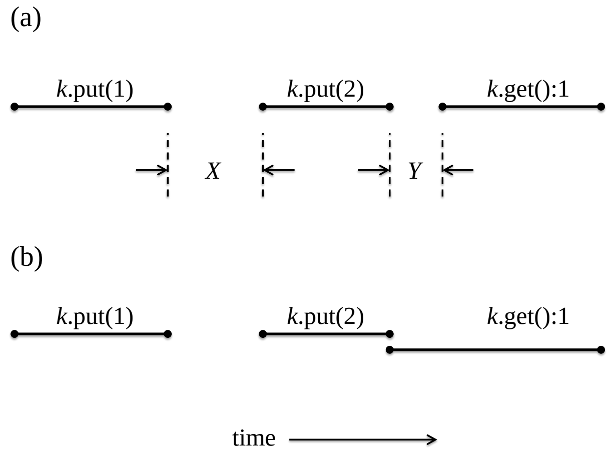


Fig. 1. Example traces illustrating Δ-atomicity, with operations represented as intervals of time. The symbols X and Y denote gaps in time between operations. The trace in part (a) is non-atomic but can be made atomic by stretching the left endpoint of the get operation as shown in part (b).

B. A new metric: Γ

The Δ-atomicity property allows get operations to return values that are stale by up to Δ time units. However, it fails to capture the idea that put operations may take effect in an order different from their "happens before" order in a trace. Consider again the example in Figure 1 (a), where k.put(1) appears to happen before k.put(2) based on the trace. It is possible that k.put(1) is ordered by the storage system as taking effect after k.put(2), making 1 the latest value instead of 2. For example, this can happen if (i) clocks are skewed across storage servers by an amount comparable to operation latencies, or (ii) writes are applied asynchronously, thereby taking effect after the end of the operation time intervals observed at clients. In such cases, the return value of the get operation is not actually stale, even though the trace appears to contain a consistency anomaly. Without knowing the exact details of every operation, we are left to interpret the causes of this anomaly as either reordering of put operations, or a get that returns a stale value.

Although both of the above interpretations are plausible, we argue on philosophical principles (inspired by *Occam's razor*) that the more likely interpretation is the one that entails less reordering. To quantify the degree of reordering, we must consider both how stale the get operation is, and how badly out of order the two puts are. As described above in the discussion of Δ, the staleness of the get operation is Y since the trace can be made atomic by stretching the get by Y time units. Alternatively, we can make the trace atomic by shifting the end time of k.put(1) to the right, or the start time of k.put(2) to the left, to close the gap of X time units between the two puts shown in Figure 1 (a). In that case, k.put(1) no longer happens before k.put(2), and so the get is no longer stale. Thus, in some sense the two puts are out of order by X time units. The quantity $\min(X, Y)$ considers all possible causes of the observed consistency anomaly in this case, and adopts the cause indicating the least reordering as the most likely one. This idea defines our new metric Γ.

To illustrate Γ in action, Figure 2 presents an additional example for comparison against Figure 1 (a). In both figures, X denotes the shortest distance by which the two puts can be considered out of order, and Y denotes the shortest distance by which the get operation can be considered stale. In Figure 1 (a), $X > Y$ holds and so we interpret the situation as a stale read. In Figure 2, $Y > X$ holds, and so we consider the two puts out-of-order. The latter trace is Γ-atomic for any $\Gamma \geq X$, and Δ-atomic only for any $\Delta \geq Y > X$.

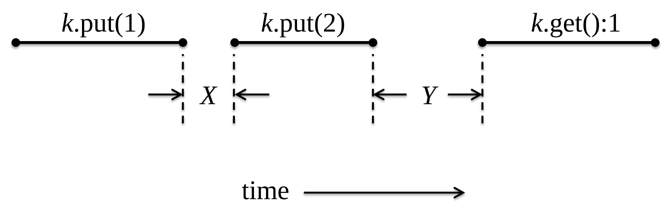


Fig. 2. Example of a three-operation trace that is Γ-atomic but not atomic, and for which Γ is smaller than Δ.

We now give a general definition of Γ: an execution trace T is Γ*-atomic* if T can be made atomic by shifting the start time of each operation (i.e., get *or* put) to the left by $\Gamma/2$ time units, and by shifting the end time of each operation to the right by $\Gamma/2$ time units. The intuition behind shifting the endpoints by $\Gamma/2$ (as opposed to Γ) is to close the gap between a pair of operations from both sides, for example stretching put(1) and put(2) in Figure 2 by $X/2$ time units each. As we explain in Section III-D, this allows us to compute Γ for some traces where Δ is undefined, such as when clock skew re-orders gets and puts. Treating start and end time points symmetrically also avoids awkward corner cases in the definition of Γ.

Like Δ, the Γ metric can be defined at various granularities. The most fine-grained method, illustrated in Figure 1, captures the inconsistency caused by the interaction of operations that access two different values, say v and v', for the same key k (e. g. $v = 1$ and $v' = 2$ in Figure 1). We call this quantity the $\Gamma(k, v, v', T)$ *score* for a trace T. Next, we construct a per-key quantity called the $\Gamma(k, T)$ *score*:

$$\Gamma(k,T) = \max_{\text{values } v,\, v' \text{ accessed on } k \text{ in } T} \Gamma(k, v, v', T)$$

Finally, we can compute a per-trace metric called the $\Gamma(T)$ *score* as follows:

$$\Gamma(T) = \max_{\text{key } k \text{ accessed in } T} \Gamma(k, T)$$

Under the assumptions described shortly, it is straightforward to show that $\Gamma(T)$ is the smallest real number x such that T is Γ-atomic for $\Gamma = x$.

To simplify our discussion of the assumptions underlying Γ, we begin with a definition: for any trace T, any key k, and any value v, let a *dictating put* for a get of k that returns v be any put in T that assigns v to k. Now it follows that $\Gamma(T)$ is well-defined (i.e., it exists and is finite) under the following assumption:

Assumption 1: Every get in T has at least one dictating put.

Intuitively, under Assumption 1 a get operation can only return data written by some put operation in the trace. In that case, an atomicity violation can occur only if some operations appear to take effect in an order that contradicts the "happens before" partial order. For example, a get may appear to return a stale value, as in Figure 2, or a get may precede all of its dictating puts in the trace due to clock skew. Shifting both the start and end point of every operation by Γ/2 time units ensures that for some large enough Γ, all operations are stretched to the point that they overlap at a common point in time. In other words, the "happens before" partial order becomes trivial, and hence the stretched trace becomes atomic. Thus, every trace satisfying Assumption 1 is Γ-atomic for some large enough Γ.

In practice, gets applied to a key-value storage system may on rare occasion return "junk" data in violation of Assumption 1. This behavior might occur as a result of software bugs or corrupted storage media. Our Γ metric is undefined for such traces, and we do not consider them any further.

C. Comparison of Δ *and* Γ

We discuss two differences between Δ and Γ in practice. First, we compare the two metrics empirically. To that end, Figure 3 presents plots of $\Gamma(k, v, v', T)$ and $\Delta(k, v, v', T)$ scores for an experiment using the Cassandra [14] key-value store. Only positive scores are considered since we are interested in consistency anomalies; all other scores are zero because the Γ score function is by definition non-negative. For each metric, the box plot presents the minimum, 25th percentile, 75th percentile, and maximum score. The raw scores are plotted immediately to the right of each box plot. (See Section IV for details of the experimental testbed.)

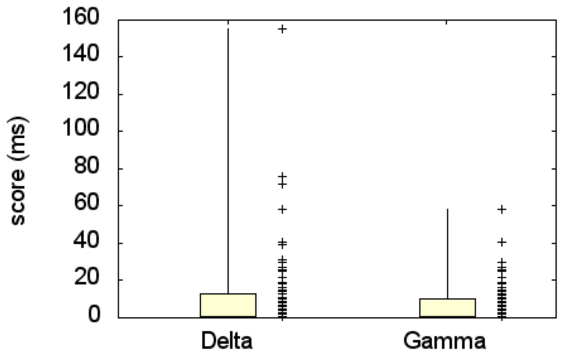


Fig. 3. Comparison of Γ and Δ scores.

The vast majority of Δ and Γ scores in Figure 3 are in the 0-30ms range. Among 213 Δ scores, 15 ($\approx$7%) exceed 30ms, and out of 203 Γ scores only 3 ($\approx$1.5%) exceed 30ms. The maximum Δ score is 155, and the maximum Γ score is 58, roughly 62% lower. Thus, the Γ scores appear less noisy than the Δ scores in the sense that the distribution of Γ scores has a shorter tail.

Further to our informal observation that Γ tends to yield less noisy metric values than Δ, we claim specifically that Γ is more robust than Δ against clock skew. This claim is based on theory rather than empirical evidence: badly skewed clocks can break a fundamental assumption that underlies the definition of Δ but is not required for Γ. As a result, there exist traces for which Γ is well-defined but Δ is undefined. Specifically, this occurs when every get has a dictating put but some get appears to happen before any of its dictating puts in a trace. As an example, consider a trace T in which a client gets a value v from some key k, and later on some other client writes v to key k. Thus, the get operation appears to return a future value in the trace. As a result $\Delta(T)$ does not exist because stretching the left endpoint of the get cannot make the trace atomic by creating overlap with a dictating put. In contrast, $\Gamma(T)$ remains well-defined since Assumption 1 holds.

We observed the above phenomenon quite frequently in runs obtained using our experimental platform, where clock skew across machines is comparable to operation latencies. Our execution traces are generated by merging timing information from multiple clients, and so the order of operations in a trace may not reflect accurately the true "happens before" partial order. Thus, gets may appear to precede their dictating puts, and pairs of puts may appear out of order in the trace. The Γ metric is designed to account for both types of anomalies.

D. *Efficient computation of* Γ

Having defined the Γ metric and its properties, we now describe an efficient (i.e., polynomial-time) algorithm that calculates $\Gamma(T)$ given an arbitrary execution trace T. Our algorithm relies on Assumption 1, which ensures that $\Gamma(T)$ is well-defined. In addition, we require the following assumption to circumvent the NP-completeness result discussed in Section II:

Assumption 2: If two puts occur on the same key, they assign distinct values.

Assumptions 1 and 2 together imply that every get has a uniquely-defined dictating put. Also note that Assumption 2 does not incur any loss of generality because it can be enforced in practice by incorporating a unique identifier (e.g., client ID and timestamp) into the value written by a put. In practice, enforcing Assumption 2 might result in Γ taking on somewhat pessimistic values. For example, if a client repeatedly puts and gets the same value in a given key, then each put is considered unique under Assumption 2, and so a positive Γ score may occur even though the client does not observe any consistency anomaly. In our experience, such workloads are rare.

The algorithm for computing $\Gamma(T)$ is inspired by ideas from the atomicity verification algorithm of Gibbons and Korach [26], and the algorithm for computing $\Delta(T)$ [8]. We first review some important technical definitions. A *cluster* is a set of operations that act on the same key, and get or put the same value. In this paper, the cluster of operations for key k and value v is denoted $C(k, v)$. For example, Figure 2 had two clusters: $C(k, 1)$ and $C(k, 2)$. Cluster $C(k, 1)$ comprises put(1) and the get operation, while cluster $C(k, 2)$ comprises only put(2). Note that under Assumptions 1 and 2, every cluster contains exactly one put and zero or more gets.

Corresponding to each cluster $C(k, v)$, we define a *zone* denoted $Z(k, v)$. The zone is an interval of time during which the value v is considered the latest value. Intuitively, overlaps among zones represent consistency anomalies, although as we explain later on some of these overlaps are benign. Formally, $Z(k, v)$ is the time interval from the earliest end time of an operation in $C(k, v)$ (denoted $Z(k, v)_{\min}$) to the latest start time of an operation in $C(k, v)$ (denoted $Z(k, v)_{\max}$). A zone $Z(k, v)$ has a well-defined *direction*: it is a *forward zone* if $Z(k, v)_{\min} \leq Z(k, v)_{\max}$, and we call it a *backward zone* otherwise. According to this definition, a backward zone ends before it starts.

We observe that a backward zone occurs when all the operations in a cluster overlap at a common point. When we discuss overlaps among zones, we treat a forward zone $Z(k, v)$ as the closed interval $[Z(k, v)_{\min}, Z(k, v)_{\max}]$, and a backward as the closed interval $[Z(k, v)_{\max}, Z(k, v)_{\min}]$. Examples of zones are shown in Figure 4, and we refer to them while discussing the algorithm.

The key insight underlying the algorithm for computing $\Gamma(T)$ is that it suffices to consider pairs of clusters individually. The input T is processed in a series of phases that identify clusters and zones, and then score the consistency anomalies caused by conflicts among clusters. For each pair of clusters, the score indicates how far the operations in those clusters must be stretched to eliminate the conflict, which depends upon the pattern of overlap among the time intervals representing the corresponding zones. Formally, we say that two forward zones conflict if their intervals overlap, and a forward zone conflicts with a backward zone if the forward interval encloses the backward interval entirely [26].

The effect of shifting operation start and end points in a given cluster by Γ/2 time units is to either shrink or expand the corresponding zone by Γ time units. Backward zones expand, whereas forward zones shrink and eventually turn

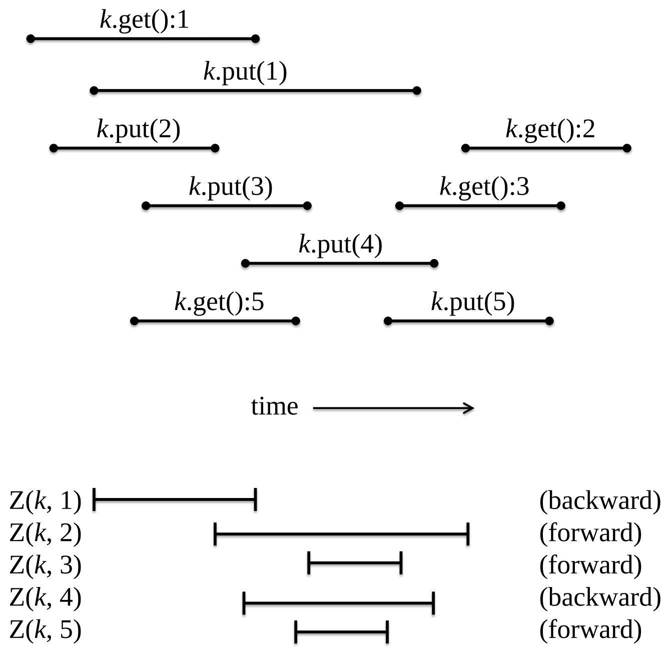


Fig. 4. Examples of forward and backward zones.

into backward zones. In contrast, in the computation of Δ the effect of shifting only the start points of get operations is more complex. For example, if a backward zone contains one put and no gets, then the zone is unaffected by the shift. For this reason, computing Γ is simpler than computing Δ.

Phase 1: Organize operations in T into clusters, and identify the zone for each cluster.

Phase 2: For any key k accessed in T, and for any pair of values v, v' written to k in T, define $\Gamma(k, v, v', T)$ as follows:

- If $v = v'$ (i.e., $Z(k, v)$ is compared against itself), assign $\Gamma(k, v, v', T) = 0$ unless some get of v precedes the put of v in cluster $C(k, v)$. In the latter case, $\Gamma(k, v, v', T)$ is the distance from the earliest finish time of a get in $C(k, v)$ to the start of the dictating put. Such a zone could arise due to clock skew. See for example zone $Z(k, 5)$ in Figure 4.
- Else if $Z(k, v)$ and $Z(k, v')$ are distinct conflicting zones (i.e., forward overlaps forward, or forward encloses backward) then $\Gamma(k, v, v', T)$ is defined as the minimum of $Z(k, v)_{\max} - Z(k, v')_{\min}$ and $Z(k, v')_{\max} - Z(k, v)_{\min}$. Intuitively, to remove the conflict, forward zones must be contracted and backward zones expanded so that the "max" endpoint of one zone moves past the "min" endpoint of the other. Note that if $Z(k, v)$ and $Z(k, v')$ are both forward zones then the shorter of the two may become a backward zone as its operations are stretched.

  This case could arise naturally when gets are applied shortly after puts, before the system has had a chance to synchronize replicas. See for example zone pairs $Z(k, 2)$ & $Z(k, 3)$, $Z(k, 3)$ & $Z(k, 5)$, as well as $Z(k, 2)$ & $Z(k, 4)$ in Figure 4.
- Else $Z(k, v)$ and $Z(k, v')$ are distinct and non-conflicting, hence $\Gamma(k, v, v', T) = 0$. In Figure 4, this case corresponds to the following combinations of zones: $Z(k, 1)$ & $Z(k, 2)$, $Z(k, 3)$ & $Z(k, 4)$, $Z(k, 1)$ & $Z(k, 4)$. This case also applies when $Z(k, v)$ and $Z(k, v')$ do not overlap.

Phase 3: For each key k accessed in T, compute $\Gamma(k, T) = \max_{v,v'} \Gamma(k, v, v', T)$, and $\Gamma(T) = \max_k \Gamma(k, T)$.

IV. Experimental Results

In this section we present the results of our experimental study of the sensitivity of the Γ metric under a variety of configuration and workload parameters. We evaluated Γ using Cassandra, a popular open-source key value store [14]. Our results demonstrate the impact on client-side consistency due to (i) critical Cassandra parameters including replication factor and read/write consistency level, and (ii) client workload parameters: get/put ratio and client throughput. Even though we only focus on failure-free executions, the experiments generated interesting and sometimes counter-intuitive results.

A secondary goal of the experimental study is to compare our Γ-based passive measurement method from prior techniques for consistency benchmarking that employ active measurement. For example, systems like YCSB++ [5] measure client-centric consistency indirectly by observing the "time lag" between a put and the first get that returns the value written. Our experimental hypothesis is that this time lag is a pessimistic estimate of the consistency actually observed by clients because it is obtained by probing the system in a manner that elicits consistency anomalies. In contrast, Γ captures client-perceived consistency directly, based on passive calculations. Because the time lag and Γ are both expressed in units of time, we can compare these two metrics directly.

Since the YCSB++ code base is not integrated into the main YCSB branch and does not support connectors suitable for the version of Cassandra used in our experiments, we rely on an alternative technique to estimate the time lag metric. Namely, we quantify the length of time needed to synchronize replicas of a key following a put by injecting a small number of additional puts into the workload using write-ALL consistency, and measuring their latency. These additional puts are executed 500ms apart in parallel with the YCSB workload, but do not participate in the calculation of Γ. Because the additional put operations wait for acknowledgment from all N replicas, their latency measures directly the time required to update the last replica. We expect this to be a reasonable estimate of the metric computed by YCSB++, where time lag measurements are obtained using read-ONE/write-ONE client-side consistency levels. (The ability to vary these levels is not implemented in the YCSB++ Cassandra connector as of 31/03/2014.)

A. Experimental setup

The experimental testbed comprises ten 64-bit 2.2 GHz dual-core AMD Opteron servers equipped with 7200 RPM SATA disks and Gigabit Ethernet. Five servers were used to execute the storage system (Cassandra 1.2.4), and another five to run a version of YCSB 0.1.4 modified to collect execution traces in the format required for our offline passive analysis. The software environment included CentOS 5.5 Linux and OpenJDK 1.7.0_19. Clocks on the cluster machines were synchronized using NTP, yielding 50-100ms accuracy with respect to stratum 0. However, we verified using ntpq that clocks on the cluster machines generally agreed to within a few milliseconds relative to each other. Accordingly, we estimated the margin of error in our time measurements as 10ms.

Each YCSB experiment consisted of a load phase, followed by a 60-second work phase. Unless otherwise specified, the following YCSB parameters were used: eight client threads, 128-byte values, and a get/put ratio of 0.5. In runs where the YCSB "hotspot" distribution is used, 20% of the keys received 80% of the load. The YCSB-Cassandra connector used write ONE / read ONE consistency by default, and Cassandra was configured with a replication factor of 3. The typical throughput for this setup was between 500 and 1000 operations per second per Cassandra host.

The size of the keyspace varies across experiments because, as we discovered, it has a very strong effect on consistency. In particular, the keyspace must be kept quite small so that consistency anomalies can be observed at all. In general, we used the largest possible keyspace for which we could observe anomalies involving at least 1% of the operations. Unless stated otherwise, this corresponds to 1000 keys for the "hotspot" distribution, and 10000 keys for the "latest" distribution, which is similar to Zipfian but skewed further toward recently inserted records [29]. Any graphs that display data for both hotspot and latest distributions are intended to demonstrate the sensitivity of Γ rather than to enable side-by-side comparisons of the two distributions.

B. Visualization of Γ

Quantifying eventual consistency precisely is challenging because the ramifications of sacrificing strong consistency cannot be sufficiently characterized by a single metric. As a result, in this paper we consider separately the frequency and severity of consistency anomalies.

1) Frequency: Using the Γ metric we devise a formula that counts the proportion of values involved in consistency anomalies, which is one way to interpret the frequency of consistency violations. The formula is expressed in terms of a trace T and Γ scores among pairs of clusters:

$$\frac{|\{(k,v) \mid \Gamma(k,v,v',T) > 0 \text{ for some } v'\}|}{\#\text{ of } (k,v) \text{ such that an op. on } k \text{ accesses value } v \text{ in } T}$$

We visualize such proportions using bar graphs, which include error bars representing the standard error calculated as $\sqrt{\hat{p}(1-\hat{p})/n}$. Here $\hat{p}$ denotes the proportion obtained using our formula (i.e., the sample proportion), and n is the denominator in the formula (i.e., the sample size).

2) Severity: To quantify the severity of consistency anomalies captured by the Γ metric, we visualize $\Gamma(k,v,v',T)$ scores using box plots, similar to Figure 3. The boxes indicate the 25th and 75th percentile scores, whereas the top and bottom lines indicate the minimum and maximum. The minimum (and often the 25th percentile) in our experiments is typically 1ms, corresponding to the smallest positive score that can be observed using the millisecond precision clock provided in Java. (Although Java does support a nanosecond-precision clock, it can only be used to measure elapsed time, whereas passive analysis is based upon wall clock time.)

Our measurements of system-centric consistency or "time lag", which are based upon the latencies of write operations, provide an approximate upper bound for the the actual severity calculated using Γ. We attempted to visualize the system-centric numbers using box plots, like Γ, but found that the distributions were strongly skewed toward small values (i.e., 2-3 ms), making the 25th and 75th percentiles difficult to separate visually from the minimum. Instead we present system-centric consistency using one-dimensional scatter plots, which show the largest measurements directly.

C. Sensitivity analysis

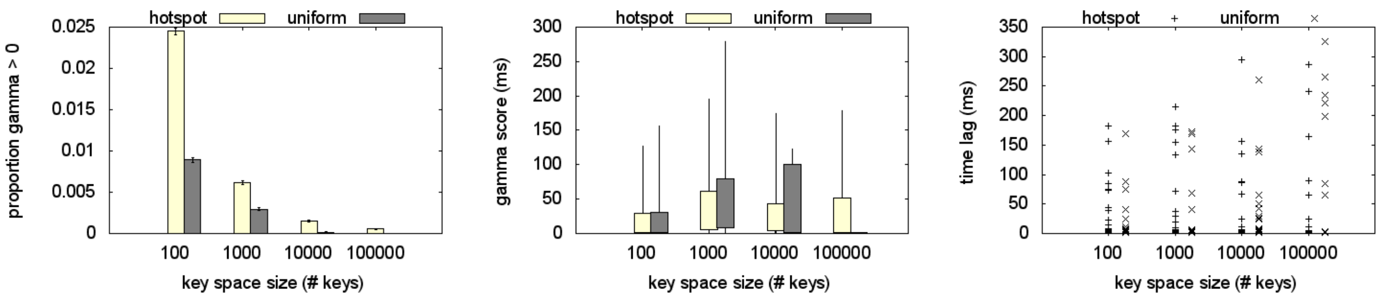
In this section we investigate the sensitivity of Γ, and the time lag metric. For each experiment, we visualize consistency using the three types of plots described earlier in Section IV-B. For example, in Figure 5 part (a) shows the proportion of values with positive $\Gamma(k,v,v',T)$ scores, part (b) is a box plot showing quartiles of the $\Gamma(k,v,v',T)$ scores, and part (c) is a scatter plot of the time lag values.

1) Sensitivity to keyspace size: Figure 5 compares runs using the YCSB hotspot and uniform distributions for various keyspace sizes. Two patterns can be seen from the results. First, the proportion of positive Γ scores decreases as the size of the keyspace increases. Second, the proportions are higher for the hotspot distribution, which is skewed toward a subset of the keyspace, than for the uniform distribution. Both trends can be explained by considering the throughput per key, which affects the likelihood of gets occurring shortly after their dictating puts. As the keyspace size increases, the throughput per key decreases; hence consistency improves. Skewing the workload has the opposite effect on the subset of "hot" keys.

The results also show that time lag is more pessimistic than Γ. The time lag values for the uniform distribution over 100,000 keys suggest inconsistencies in the hundreds of milliseconds. However there were no Γ scores above the margin of error of around 10ms in the uniform/100,000 case. As a result, there is no evidence of clients observing any consistency anomalies in this particular run. Γ is thus more accurate as a measure of observed inconsistency.

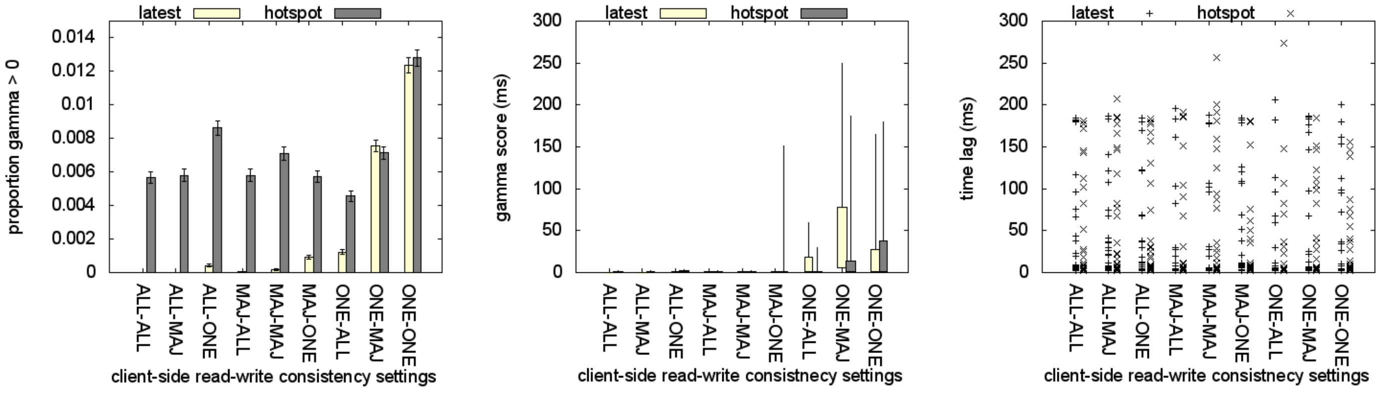
2) Sensitivity to client-side consistency settings: Figure 6 compares runs with various combinations of client-side read and write consistency levels, using the latest (10,000 keys) and the hotspot (1,000 keys) distributions, and a replication factor of 5. The proportion of positive Γ is highest when the weakest settings (i.e., ONE-ONE) are used. The box plot shows a definitive pattern: Γ scores are substantially worse for ONE-*, otherwise the scores are generally in the 1-2ms range (except for MAJORITY-ONE), which is within the 10ms margin of error. In contrast, the time lag does not show any variation because it is always measured using put operations with ALL consistency.

The box plot for the ONE-ALL run indicates consistency violations substantially higher than the 10ms margin of error. This result demonstrates that atomicity is in fact stronger than the popular notion of "strong consistency" [2], which holds in this experiment because reads and writes access overlapping subsets of replicas. That is, under ONE-ALL consistency the criterion $R + W > N$ is satisfied with $R = 1$ and $W = N$, where R is the number of replicas accessed by a read and W



(a) frequency (b) severity (c) system-centric consistency

Fig. 5. Consistency versus keyspace size.



(a) frequency (b) severity (c) system-centric consistency

Fig. 6. Consistency versus number of replicas read and written. (E.g., "ONE-MAJ" indicates read one and write majority.)

is the number of replicas accessed by a write. We comment on this case in more detail below.

Strong consistency is usually discussed in the special case when gets do not overlap in time with puts, and puts take effect in a well-defined order. Under these simplifying assumptions, the $R + W > N$ rule indeed guarantees that a get operation always returns the latest value. However, the same rule does not guarantee atomicity in the general case when gets and puts may overlap arbitrarily. Attiya, Bar-Noy and Dolev observe this issue in their simulation of shared memory on top of message passing, and use an explicit write-back phase to ensure that when one read observes a partially completed write, all future reads observe either the same write or a later one [30]. In contrast, Cassandra lacks a write-back phase and hence may exhibit atomicity violations even under conditions that guarantee strong consistency.

To illustrate the above point, we analyzed the operations corresponding to the maximum Γ score in the ONE-ALL run in Figure 6 for the "latest" distribution. A simplified illustration of these operations is shown in Figure 7, which includes two gets and two puts. The first get operation observes a partially completed put of value 2, but the second get reaches a replica that still holds the older value 1. This constitutes a violation of atomicity because the "happens before" order requires k.put(2) to take effect after k.put(1), and yet 2 is read from k before 1. This type of anomaly can only be observed when gets execute concurrently with puts—a scenario that falls outside the scope of both active measurement and PBS (see Section II).

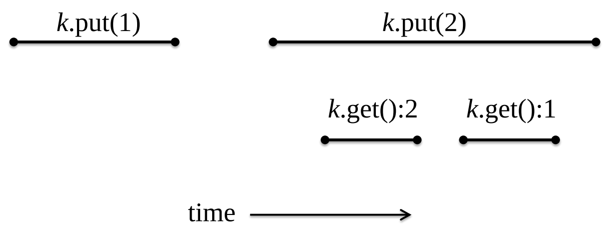
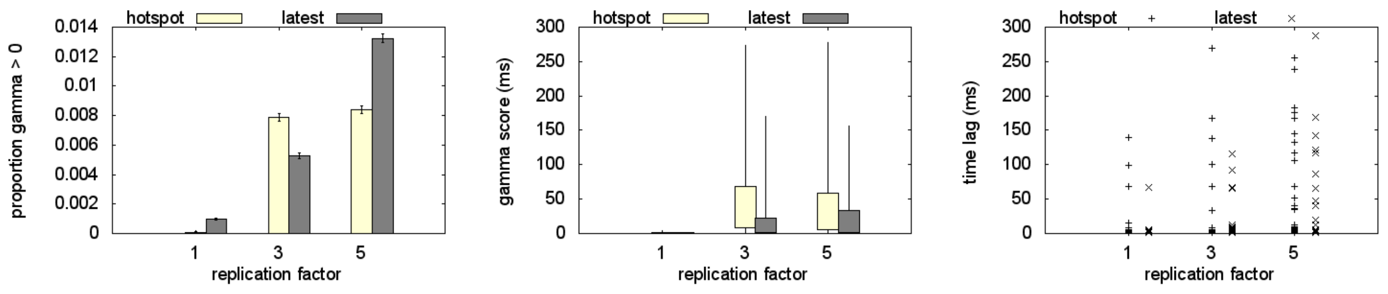


Fig. 7. Simplified illustration of an atomicity violation occurring despite overlapping read and write quorums.

3) Sensitivity to replication factor: Figure 8 compares runs with various replication factors, using the hotspot (1000 keys) and latest (10000 keys) distributions. Both the frequency and severity of consistency violations (Figures 8 (a) and (b)) are much greater when the replication factor is 3 or 5, versus 1. This is expected since we used the weakest consistency settings (read-ONE/write-ONE), which ensure overlapping read and write quorums only when the replication factor equals one. In comparison, the time lag plot exhibits only a weak trend of increasing values as the replication factor grows, with a very small difference between 1 and 3.

We observe a few positive Γ scores even with a replication factor of 1. Specifically, out of approximately 100-120k values written to the storage system, fewer than 100 resulted in consistency violations. The box plot reveals that the Γ scores



(a) frequency (b) severity (c) system-centric consistency

Fig. 8. Consistency versus replication factor.

in that case were in the 1-2ms range, well within the 10ms margin of error.

4) Sensitivity to get/put ratio: Figure 9 compares runs with various proportions of gets for the latest and hotspot distributions over 100 keys. A very small keyspace was chosen for this particular experiment to demonstrate the sensitivity more clearly, and the total throughput was approximately constant across runs. The proportion of positive Γ scores exhibits a definitive growth as the fraction of puts increases. In contrast, the 75th percentile Γ score shown in the box plot tends to decrease slightly. These outcomes can be explained as follows: a higher proportion of gets increases the likelihood that clients observe consistency anomalies (e.g., by reading a key shortly after it is written), but a correspondingly lower proportion of puts decreases the pressure on the replication protocol, and hence the severity of observed anomalies declines.

5) Sensitivity to throughput: Figure 10 compares runs at various levels of throughput for the "latest" distribution over 10000 keys. The proportion of positive Γ scores increases steadily with throughput, which is expected since the frequency of operations per key increases as well. In comparison, the box plot and time lag plot do not exhibit a clear pattern. The maximum time lag seems to increase slightly with throughput, which is expected given the additional load on the replication protocol.

V. Conclusion

In this paper, we presented a theoretically-sound methodology for benchmarking eventual consistency, and demonstrated its use on Cassandra. We showed experimentally that our client-centric technique exhibits substantial sensitivities to a variety of configuration and workload parameters, most importantly the keyspace size, client-side read/write consistency levels, distribution of keys, and replication factor. In comparison, measurements of a system-centric consistency (i.e.,"time lag") metric were shown to exhibit weak-to-no sensitivity, and indicated that substantial consistency anomalies might occur even in cases when no such anomalies were observed using our metric Γ.

Our consistency benchmarking methodology leads to many interesting future directions. A challenge with using a prediction tool like PBS [11] is developing an accurate system model, particularly for modeling uncommon events like failures (both permanent and transient) or flash crowds. Our measurement technique provides a way to validate PBS predictions in arbitrary workload traces, which may be of particular importance when comparing the impact of uncommon events on data staleness. Γ may also provide a way to methodically tune the trade-off between consistency and latency. Consider the possibility of "amplifying consistency" by delaying or slowing down operations in a manner that improves consistency at the cost of latency. The Γ metric can guide us toward finding the optimal distribution of delays and deciding which operations should be delayed to strike a good balance. We also plan to investigate novel computation techniques for large traces. For example, for online processing we can work with a fixed size window of operations in the trace, and continuously discard old operations that can no longer affect the computation [8]. Second, we can truncate the trace and approximate Γ over the most recent operations. Third, we can parallelize the Γ computation code to exploit modern multi-core architectures. Finally we hope to study the effect of failures (e.g., network partitions) on consistency using the Γ metric.

Acknowledgments

We are grateful to Jay J. Wylie for stimulating discussions on the topic of consistency, and for his insightful feedback on earlier drafts of this manuscript. Thanks also to Doug Terry, Xiaozhou (Steve) Li, and the anonymous referees for their helpful comments. We are indebted to Prof. Ashraf Aboulnaga for providing access to a computing cluster in support of the experiments presented in this paper.

References

[1] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 1995, pp. 172–182.

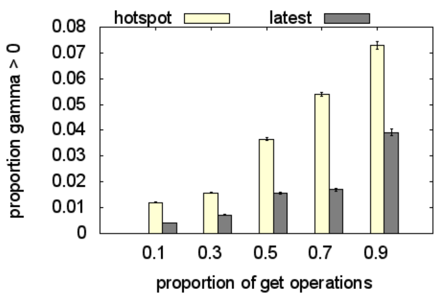
[2] W. Vogels, "Eventually consistent," *Queue*, pp. 14–19, 2008.

[3] E. A. Brewer, "Towards robust distributed systems (Invited Talk)," in *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2000.

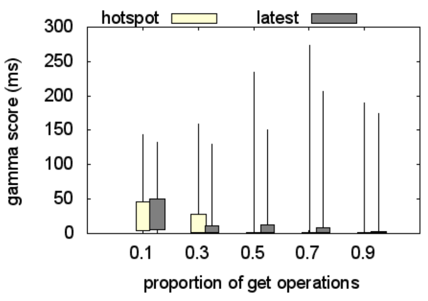
[4] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *IEEE Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[5] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," in *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2011, pp. 9:1–9:14.

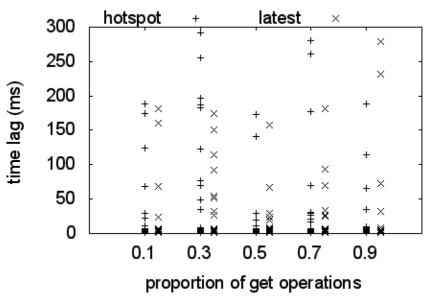
[6] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior," in *Proc. Workshop on Middleware for Service Oriented Computing (MW4SOC)*, 2011.



(a) frequency

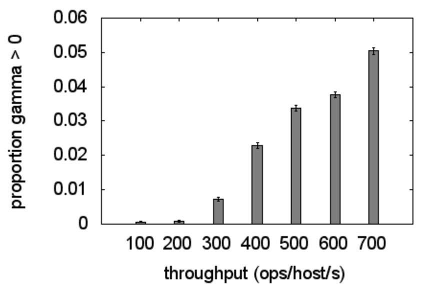


(b) severity

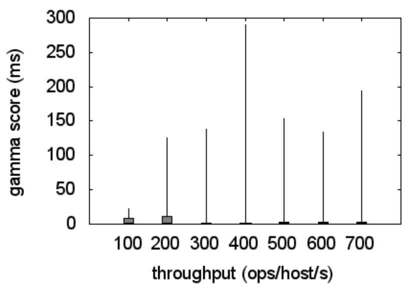


(c) system-centric consistency

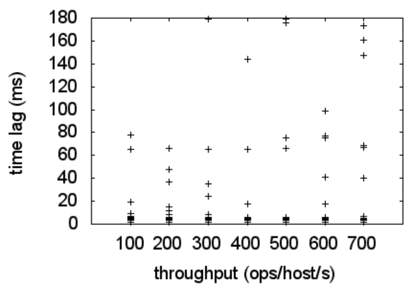
Fig. 9. Consistency versus proportion of get operations.



(a) frequency



(b) severity



(c) system-centric consistency

Fig. 10. Consistency versus throughput.

[7] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 134–143.

[8] W. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2011, pp. 197–206.

[9] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, "Toward a principled framework for benchmarking consistency," in *Proc. USENIX Workshop on Hot Topics in System Dependability (HotDep)*, 2012.

[10] A. Aiyer, L. Alvisi, and R. A. Bazzi, "On the availability of non-strict quorum systems," in *Proc. International Symposium on Distributed Computing (DISC)*, 2005, pp. 48–62.

[11] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 776–787, 2012.

[12] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," *Microsoft Research Technical Report MSR-TR-2013-39*, March 2013.

[13] Y. Zhu and J. Wang, "Client-centric consistency formalization and verification for system with large-scale distributed data storage," *Future Gener. Comput. Syst.*, vol. 26, no. 8, pp. 1180–1188, 2010.

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

[15] L. Lamport, "On interprocess communication, Part I: Basic formalism and Part II: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 77–101, 1986.

[16] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[18] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[19] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 251–264.

[20] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 309–324.

[21] W. Vogels, "Amazon's dynamo," *All Things Distributed*, 2007, http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

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

[23] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proc. International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2011, pp. 386–400.

[24] ——, "Convergent and commutative replicated data types," *Bulletin of the EATCS*, vol. 104, pp. 67–88, 2011.

[25] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 239–282, Aug. 2002.

[26] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, pp. 1208–1244, 1997.

[27] K. Zellag and B. Kemme, "How consistent is your cloud application?" in *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2012, pp. 1–14.

[28] A. Davidson, A. Rubinstein, A. Todi, P. Bailis, and S. Venkataraman, "Adaptive hybrid quorums in practical settings," UC Berkley, Tech. Rep., 2013.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2010, pp. 143–154.

[30] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995.