Toward a Principled Framework for Benchmarking Consistency

Muntasir Raihan Rahman
HP Labs, Palo Alto /
University of Illinois at Urbana Champaign
mrahman2@illinois.edu

Wojciech Golab, Alvin AuYoung
Kimberly Keeton, Jay J. Wylie
HP Labs, Palo Alto
firstname.lastname@hp.com

Abstract

Large-scale key-value storage systems sacrifice consistency in the interest of dependability (i.e., partition-tolerance and availability), as well as performance (i.e., latency). Such systems provide eventual consistency, which—to this point—has been difficult to quantify in real systems. Given the many implementations and deployments of eventually-consistent systems (e.g., NoSQL systems), attempts have been made to measure this consistency empirically, but they suffer from important drawbacks. For example, state-of-the art consistency benchmarks exercise the system only in restricted ways and disrupt the workload, which limits their accuracy.

In this paper, we take the position that a consistency benchmark should paint a comprehensive picture of the relationship between the storage system under consideration, the workload, the pattern of failures, and the consistency observed by clients. To illustrate our point, we first survey prior efforts to quantify eventual consistency. We then present a benchmarking technique that overcomes the shortcomings of existing techniques to measure the consistency observed by clients as they execute the workload under consideration. This method is versatile and minimally disruptive to the system under test. As a proof of concept, we demonstrate this tool on Cassandra.

1 Introduction

Large-scale key-value storage systems are quickly becoming an essential component of many IT infrastructures. From fast-growing start-ups to large enterprises, these systems are becoming commonplace in production use because of their ability to scale easily and the availability of many widely-supported software implementations. However, in order to provide performance and dependability at scale, the common principle followed by these key-value systems is to relax data consistency [25].

As these systems find their way into a wider variety of industries, it becomes increasingly important to understand the implications of this relaxed consistency model: to what extent relaxation improves system performance and to what extent it degrades data consistency.

For example, Web-based applications rely on key-value systems to provide high-throughput and low-latency access to content. While these applications do not strictly require serializability for correct operation, they may require a stronger property than eventual consistency, such as causal or "causal+" [16] consistency, in order to improve the user experience.

On the other hand, cloud-based health care applications likely value predictable consistency over performance. Eventually consistent updates to a patient's record may introduce mistakes along the path of patient care. For example, stale information (e.g., due to weak consistency) about a patient's dosage or medical history may lead to incorrect, or—in an extreme case—harmful treatment plans.

Today, cloud customers who care about consistency have limited means to understand or control data consistency when choosing among available storage systems, or their configurations. For example, decisions to tune "knobs" such as the replication factor or quorum size remain ad-hoc, and may lead to excessive replication or operational costs. More importantly, no combination of these knob settings can ensure that the storage system is strongly consistent (e.g., always returns the freshest data). This shortcoming is a fundamental limitation of such always-available, partition-tolerant systems, as observed by Brewer [8] and formalized by Lynch et al. [17]. Moreover, many modern systems often choose to further sacrifice consistency for better performance [4].

We argue that a methodology for comprehensive consistency measurement is necessary to evaluate today's eventually consistent systems. Such a measurement framework can identify the shortcomings of architectural designs or implementation errors in existing systems. Moreover, it can determine the actual consistency behavior of a particular deployment, which may be helpful to guide configuration and deployment decisions.

Prior techniques for measuring consistency follow a methodology that is oversimplified, and as a result suffer from important drawbacks. For example, the act of measurement disrupts the workload by injecting operations, causing a troublesome "observer effect". Moreover, the injected operations tend to stress the system, which may elicit worst-case behavior even for a light workload. Understanding observed, as opposed to worst-case, consistency is important for systems designers considering performance trade-offs, particularly if observed consistency is vastly different from the worst-case.

Our position is simple—a consistency benchmark should produce precise and accurate measurements of consistency with minimal disruption to the system under evaluation. These measurements should reflect the consistency actually observed by clients in the workload under consideration, rather than the consistency of operations injected artificially into the workload. Furthermore, a benchmark must collect measurements in a system-agnostic way, enabling comparisons not only between different implementations of the *same* consistency model (e.g., sloppy quorums [24]), but also between different consistency models.

In this paper, we describe a principled approach to consistency measurement that captures more faithfully and accurately the actual consistency behavior of a key-value storage system for an arbitrary workload. Our specific contributions are:

1. A survey of known techniques for quantifying and benchmarking consistency, and discussion of their limitations (Section 2).
2. An outline of a more general and precise approach to consistency measurement (Section 3).
3. A proof-of-concept benchmarking tool, which we use to obtain consistency measurements for the Cassandra [1] key-value store (Section 4).

2 Related work

Consistency in this paper refers to the notion that different clients accessing a storage system agree in some way on the state of data. In the literature, this is termed the *client-centric* view, as opposed to the *data-centric* view, which refers to details that are not directly observable by clients (e.g., messages in flight, state of replicas). The client-centric view is more natural in the context of benchmarking consistency, as it does not require system-specific and disruptive instrumentation to collect intimate details of the execution. Instead, it considers only the information that clients can capture locally as they apply *get* and *put* operations on keys, such as the start and end time of each operation as well as its arguments and response.

Client-centric definitions of consistency typically refer to agreement on when and in which order operations take effect (e.g., see [22]). As we discuss shortly, early attempts to benchmark consistency focus on the "when", and interpret this question as meaning roughly "How soon after a write operation returns do read operations return the written value?", or in other words, "How eventual is eventual?" [7, 19, 26].

Formalizing and answering these questions precisely bring us a step closer to understanding the complex relationship between the workload applied to a storage system, the failure pattern, the configuration parameters, and the observed client-centric consistency. In contrast, prior work covers a narrow sub-space of this multidimensional relationship that considers only failure-free executions, and relies on an informal methodology that exercises the storage system only near the limits of its "consistency envelope".

Definitions of version and time-based staleness
Staleness is a fundamental concern in data management, and can be used to describe the quality of both the data and the system that stores it. In this benchmark, we focus on the quality of the storage system, and in particular the protocol synchronizing different replicas of data. To that end, we consider staleness as a relative measure: how long ago was the value read first updated (e.g., see Figure 1). In other words, data becomes stale the first time it is overwritten by newer data.

Prior techniques for quantifying staleness in key-value storage systems either count versions (e.g., the value read is the second-latest value written) or measure time (e.g., the value read is one hour older than the latest value written) [5, 11, 14, 27, 28, 29]. These quantities are easy to state precisely under the simplifying assumption that read and write operations are instantaneous—a collection of unique points on a one-dimensional axis. In that case, there is a natural total order on the operations, and moreover the "latest value" at any point in time is well defined. In contrast, in real-world scalable storage systems, operational latencies due to processing, networking and I/O are non-trivial, and so there can be multiple operations in flight at any given time, even on a single key. Thus, non-trivial latencies and parallelism complicate reasoning about when a given operation takes effect, as well as the order in which operations take effect relative to each other.

A more precise treatment of staleness devised by the theory community includes the (client-centric) concepts of k-atomicity [5] and Δ-atomicity [11]. The k-atomicity property is a natural formalization of version-based staleness. An execution of operations in a key-value store is

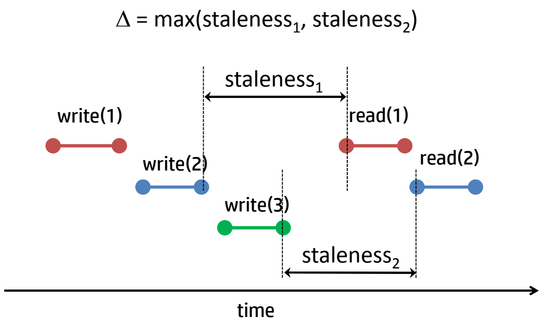

Figure 1: Example calculation of Δ.

k-atomic if the operations in that execution can be totally ordered so that: (1) the total order extends the "happens before" partial order (i.e., if operation A ended before operation B began during the execution, then A precedes B in the total order); and (2) each read returns the value assigned by one of the k most recent writes preceding the read in the total order. (In the case $k = 1$, k-atomicity corresponds to Lamport's atomicity concept [15], which we discuss below.) For any given execution, we can quantify version-based staleness by solving the following optimization problem: find the smallest k for which the execution is k-atomic. We are not aware of an efficient (i.e., poly-time) solution to this problem, although [11] presents progress toward solving the corresponding decision problem for $k = 2$.

The Δ-atomicity property attempts to capture time-based staleness by stating that read operations must return values that are at most Δ time units staler than the latest value for a key. More formally, if we "stretch" the start time of each read to a point Δ time units earlier, then the resulting execution should be atomic in Lamport's sense [15]. For any given execution, it is possible to compute the smallest $\Delta \geq 0$ for which that execution is Δ-atomic using an efficient algorithm [11].

Figure 1 illustrates Δ in action. The start and end times are shown for three writes and two reads, all operating on the same key. We assume that each operation takes effect between its beginning and end. For example, 2 is the latest value from the moment write(2) ends, and possibly even earlier. Thus, read(1) returns a value that is stale by at least the width of the "gap" between it and write(2). Even though write(3) is the latest value, staleness for read(1) is measured from the *first* unseen update to the key: write(2). Similarly, the staleness for read(2) is measured from the end of write(3).

For completeness, we also briefly discuss well-studied notions of weakly consistent shared objects from distributed computing theory literature. Lamport proposed the notions of *safe*, *regular* and *atomic registers* (i.e., shared objects that support read and write operations). These specifications describe the correct behavior of read operations when they can execute concurrently with writes and with each other, but do not adequately capture the possibility that non-concurrent operations may appear to take effect out of order—a commonplace phenomenon in modern quorum-replicated systems. Lamport's atomicity property is similar in spirit to Herlihy and Wing's *linearizability* [12] and Papadimitriou's *strict serializability* [18] for read/write register objects.

Measuring and bounding staleness

Several papers attempt to measure or bound staleness in order to characterize the spectrum of trade-offs surrounding Brewer's celebrated CAP principle [4, 8]. Wada et al. [26] measure time-based staleness in cloud storage platforms by writing timestamps to a key from one client three times per second, reading the same key from another client fifty times per second, and computing the difference between the reader's local time and the timestamp read. In experiments using Amazon's SimpleDB [3], they observe staleness on the order of seconds.

The methodology of Wada et al. is sufficient to obtain evidence relevant to their central research question—whether cloud storage systems in practice provide more consistency than they promise. However, their technique also has several disadvantages as a consequence of exercising the system in an artificial way. First, the measurement is disruptive because it introduces additional write operations to the workload. This is unsuitable in a production environment, unless the operations are applied to a special "dummy" key, in which case the outcome may not predict accurately the staleness observed by reads on the other keys. Secondly, the technique is pessimistic because it considers a pattern of access where read operations occur back-to-back with writes. This measurement captures the minimum time needed for replicas of a key-value pair to synchronize, but in a real world workload, gaps between operations may result in clients observing far less staleness. In particular, if the load is trivial then it is possible that all operations (except the ones injected artificially) will be atomic. A third drawback is the use of only a single writer. While this certainly simplifies calculations, the measurements obtained may fail to cover special execution paths of the storage system for dealing with concurrent writes, which hurts accuracy further.

Bermbach et al. [7] and Patil et al. [19] measure staleness using techniques similar to Wada et al. The latter paper presents an extension of the Yahoo Cloud Serving Benchmark (YCSB) [9], with support for basic consistency benchmarking. Their technique relies on a middleware service, namely ZooKeeper [13], to convey timing information between readers and writers. This technique is limited in precision due to the latency introduced by operations on ZooKeeper, and hence it produces results with one-sided error: reported consistency violations are true assuming synchronized clocks, but lack of reported violations does not imply atomic behavior.

Bailis et al. [6] consider the problem of predicting the staleness from an abstract model of the storage system, including details such as the distribution of latencies for network links. This work considers both version and time-based staleness, and provides an upper bound on the probability that a client observes stale data. This prediction, similar to the measurements of Wada et al., may be overly pessimistic for light workloads. Predicting and measuring staleness are complementary techniques—prediction can be used for planning and measurement can be used in a variety of ways, such as performance tuning, monitoring, evaluating service-level agreements, and feedback control.

Other work
Shapiro et al. formalize eventual consistency for shared objects that avoid conflicts by design, for example by providing commutative operations [20, 21]. Conventional key-value storage systems, like Cassandra, fall outside this category because write operations are inherently conflict-prone. Zhu et al. [30] give formal definitions of eventual consistency for read/write storage systems, as well as several client-centric properties: read-your-writes, monotonic reads, writes follow reads, and monotonic writes. This work does not provide a way to measure the difference between a particular consistency property and the actual consistency delivered by a storage system. Less formal definitions of eventual consistency appear in numerous papers (e.g., [23, 25]).

3 Toward a benchmarking framework

We focus on creating a client-centric benchmarking tool that measures observed consistency and is minimally disruptive to the system under evaluation. Since consistency and fault-tolerance are intimately related in eventually consistent systems, the tool should provide support for fault injection. This includes crashes (individual and correlated) as well as network partitions, and necessitates "white-box" access to the infrastructure. Finally, the tool should simplify analysis of the results by presenting useful visualizations to the user.

As a stepping stone towards building a comprehensive benchmarking framework, we now describe a methodology for minimally disruptive measurement of consistency in arbitrary workloads. We then suggest how such measurements might be visualized. Since our methodology is client-centric, it can be married with any workload generator. The measurement entails collecting timing information at clients for an arbitrary interleaving of operations, and calculating consistency metrics only from this information using theoretically-sound techniques. As a running example, we consider the calculation of the Δ quantity described in Section 2, and then discuss integration with YCSB [9].

Δ-atomicity is defined abstractly for arbitrary executions, including ones containing concurrent writes to the same key. To quantify staleness, we propose to calculate Δ for a given execution using the procedure described in [11]. First, we group operations into *clusters*—sets of operations that access the same key and read or write same value [10]. For example, in Figure 1 there are three clusters, red, blue and green, corresponding to the values 1, 2 and 3. Next, we choose a key k and for each pair of clusters for that key, and we determine the staleness due to the interaction of operations in these clusters by evaluating a *scoring function* χ [11]. We omit the formal details and point out only that in Figure 1, χ is the width of the staleness "gaps" experienced by $\mathsf{read}(1)$ and $\mathsf{read}(2)$. Finally, we compute the Δ value for key k by taking the maximum of χ over all pairs of clusters for k. We repeat for each key and, taking the maximum, obtain a global Δ indicating the staleness for the entire execution. Note that since the calculation combines time values from multiple hosts, accuracy is contingent upon synchronized clocks.

The quantities χ and Δ can be displayed visually in various ways. For example, using Δ's for different keys, we can plot a histogram that shows what proportion of the key space was read in a consistent manner. Or, using χ values for one key, we can plot a histogram that shows what proportion of clusters contained reads of stale values (which, in turn, estimates what proportion of reads returned stale values). We can also use a time series plot of χ to visualize the *instantaneous consistency* in an execution, which indicates the staleness of values read at different points in time. This allows us to observe how staleness varies over time (e.g., in response to load spikes or failures), information that is masked by Δ alone since it quantifies consistency for the duration of an entire execution. Note that χ and Δ, as well as the corresponding visualizations, can be obtained for a subset of the key space (e.g., chosen through random sampling).

4 Experimental evaluation

To demonstrate our benchmarking methodology, we integrated our consistency measurement technique into YCSB [9], and used the modified YCSB to measure consistency in Cassandra [1], a widely adopted key-value storage system. Our experiments use YCSB v. 0.1.4 and Cassandra v. 1.1.0.

The experimental hardware platform is a cluster of ten commodity dual-socket 6-core Xeon servers equipped

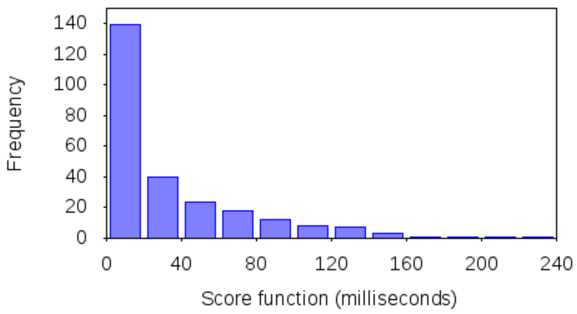

Figure 2: Histogram of score function (χ) values.

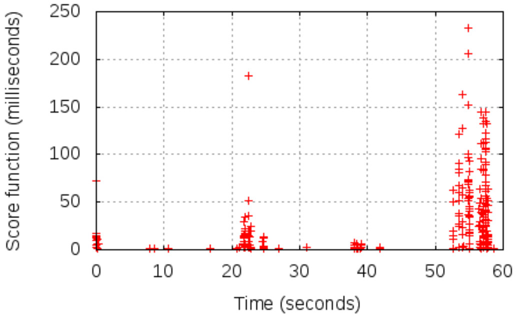

Figure 3: Time series plot of score function (χ) values.

with 1GigE network interface cards and 96GB DRAM. Each server ran a 32-thread YCSB client on one socket, and a Cassandra node on the other socket, configured with default options except as follows: keys were hashed uniformly across all nodes and 3-way replicated using the "simple" replica placement strategy [2]. By default, the Cassandra connector in YCSB used consistency level "ONE" for both reads and writes. This consistency level requires that a write be applied to the commit log and memory table of at least one replica node before returning to the client, and allows a read to return the value obtained from the first replica that responds.

We instrumented the YCSB source code to log timing information for each operation using a millisecond-precision clock. We pre-loaded Cassandra with 1000 keys and applied a read-heavy (80% get, 20% put) workload for 60 seconds. The keys were drawn from YCSB's "hot spot" distribution, with 80% of the operations going to a subset of hot keys comprising 20% of the key space.

We computed χ and Δ from collected timing information, as described in Section 3. Figure 2 is a histogram of positive χ values for all keys. Each point represents the relative staleness observed by some read operation on some key. The value of χ ranges from 1ms to 233ms, and the margin of error due to clock skew is around 1ms. In comparison, Wada et al. report much higher maximum staleness levels in their experiments using Amazon's SimpleDB (see Figures 2 and 3 in [26]).

Figure 3 shows a time series plot of the χ values for all keys). This visualization allows us to observe how staleness varies over time, in contrast to the distribution of staleness values captured in Figure 2. In Figure 3, the x-axis depicts the approximate time when a read returns a stale value, and the y-axis depicts the corresponding χ value. Most of the data points are concentrated near the x-axis, as we expect based on the histogram, and furthermore there are a few visible "inconsistency spikes".

Finally, we measured the overhead of instrumentation that is required to compute the staleness metric and observed a performance loss of less than five percent with instrumentation enabled.

5 Conclusions and future work

In this paper, we present a client-centric benchmarking methodology for understanding eventual consistency in distributed key-value storage systems. Our methodology measures observed, rather than worst-case, consistency. It extends the popular YCSB benchmark to measure the staleness of data returned by reads using the concept of Δ-atomicity [11]. Because our technique does not inject operations into the workload, it measures consistency in a more faithful manner than prior benchmarks. By measuring consistency in a system-agnostic manner, we provide a quantitative methodology for examining the performance vs. consistency trade-offs across various key-value system architectures.

Using a preliminary implementation of our methodology, we demonstrate that the staleness of data in Cassandra exhibits a long and thin tail. That is, the worst-case staleness is much higher than the typical staleness of data returned by read operations. This observation has implications for a system administrator when deciding how to configure or deploy a system like Cassandra—depending on the desired performance and deployment size, the choice of replication factor and quorum sizes can be guided by our benchmark results rather than guesswork.

We are actively extending our work to consider runs with failures. Events such as network partitions, software crashes, or device failures may trigger special execution paths in the system and result in different consistency behaviors. Our goal in future work is to stage experiments involving such failures through additional modifications to the Δ-enabled YCSB suite.

Acknowledgments Thanks to Steve Uurtamo for conducting preliminary experiments. We are also grateful to the anonymous reviewers for their helpful feedback.

References

[1] Cassandra. `http://cassandra.apache.org/`.

[2] Cassandra Wiki. `http://wiki.apache.org/cassandra/API`.

[3] SimpleDB. `http://aws.amazon.com/simpledb/`.

[4] ABADI, D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer 45*, 2 (2012), 37–42.

[5] AIYER, A., ALVISI, L., AND BAZZI, R. A. On the availability of non-strict quorum systems. In *DISC'05: Proc. of the 19th International Symposium on Distributed Computing* (2005), pp. 48–62.

[6] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 776–787.

[7] BERMBACH, D., AND TAI, S. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *MW4SOC'11: Proc. of the 6th Workshop on Middleware for Service Oriented Computing* (2011), pp. 1:1–1:6.

[8] BREWER, E. A. Towards robust distributed systems (Invited Talk). In *PODC'00: 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (2000).

[9] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *SOCC'10: Proc. of the 1st ACM Symposium on Cloud Computing* (2010), pp. 143–154.

[10] GIBBONS, P. B., AND KORACH, E. Testing shared memories. *SIAM J. Comput. 26*, 4 (1997), 1208–1244.

[11] GOLAB, W., LI, X., AND SHAH, M. A. Analyzing consistency properties for fun and profit. In *PODC'11: Proc. of the 30th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (2011), pp. 197–206.

[12] HERLIHY, M., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS 12*, 3 (July 1990), 463–492.

[13] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX ATC'10: Proc. of the 2010 USENIX Annual Technical Conference* (2010), pp. 11–25.

[14] KRISHNAMURTHY, S., SANDERS, W. H., AND CUKIER, M. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems 14* (2003), 1112–1125.

[15] LAMPORT, L. On interprocess communication, Part I: Basic formalism. *Distributed Computing 1*, 2 (1986), 77–85.

[16] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP'11: Proc. of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 401–416.

[17] LYNCH, N., AND GILBERT, S. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2 (June 2002), 51–59.

[18] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (1979), 631–653.

[19] PATIL, S., POLTE, M., REN, K., TANTISIRIROJ, W., XIAO, L., LÓPEZ, J., GIBSON, G., FUCHS, A., AND RINALDI, B. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *SOCC'11: Proc. of the 2nd ACM Symposium on Cloud Computing* (2011), pp. 9:1–9:14.

[20] SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *SSS'11: Proc. of 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems* (2011), pp. 386–400.

[21] SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. Convergent and commutative replicated data types. *Bulletin of the EATCS 104* (2011), 67–88.

[22] TERRY, D. Replicated data consistency explained through baseball. Tech. Rep. MSR-TR-2011-137, Microsoft Research, October 2011.

[23] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP'95: Proc. of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 172–182.

[24] VOGELS, W. Amazon's dynamo. *All Things Distributed* (October 2007). `http://www.allthingsdistributed.com/`.

[25] VOGELS, W. Eventually consistent. *Queue 6*, 6 (Oct. 2008), 14–19.

[26] WADA, H., FEKETE, A., ZHAO, L., LEE, K., AND LIU, A. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR'11: Proc. of the 5th Biennial Conference on Innovative Data Systems Research* (2011), pp. 134–143.

[27] YU, H., AND VAHDAT, A. The costs and limits of availability for replicated services. In *SOSP'01: Proc. of the 18th ACM Symposium on Operating Systems Principles* (2001), pp. 29–42.

[28] YU, H., AND VAHDAT, A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst. 20*, 3 (Aug. 2002), 239–282.

[29] ZHANG, C., AND ZHANG, Z. Trading replication consistency for performance and availability: an adaptive approach. In *ICDCS'03: Proc. of the 23rd International Conference on Distributed Computing Systems* (2003), pp. 687–695.

[30] ZHU, Y., AND WANG, J. Client-centric consistency formalization and verification for system with large-scale distributed data storage. *Future Gener. Comput. Syst. 26*, 8 (Oct. 2010), 1180–1188.