Fine-tuning the Consistency-Latency Trade-off in Quorum-Replicated Distributed Storage Systems

Marlon McKenzie
Electrical and Computer Engineering
University of Waterloo, Canada
m2mckenzie@uwaterloo.ca

Hua Fan
Electrical and Computer Engineering
University of Waterloo, Canada
h27fan@uwaterloo.ca

Wojciech Golab
Electrical and Computer Engineering
University of Waterloo, Canada
wgolab@uwaterloo.ca

***Abstract*—NoSQL storage systems are used extensively by web applications and provide an attractive alternative to conventional databases when the need for scalability outweighs the need for transactions. Several of these systems, notably Amazon's Dynamo and its open-source derivatives, provide quorum-based replication and present the application developer with a choice of multiple client-side "consistency levels" that determine the number of replicas accessed by reads and writes. This setting, in turn, affects both the latency and the consistency observed by the client application. Since using a fixed combination of read and write consistency levels for a given application provides only a limited number of discrete options for tuning the consistency-latency trade-off, we investigate techniques that allow more fine-grained tuning as may be required to support consistency guarantees through service level agreements (SLAs). We consider two such techniques, a novel technique that assigns the consistency level on a per-operation basis by choosing randomly between two options (e.g., weak vs. strong consistency) with a tunable probability, and a known technique that uses weak consistency and injects delays into storage operations artificially. We compare and contrast these two techniques experimentally against each other and against combinations of fixed consistency levels using Apache Cassandra deployed in Amazon's EC2 environment.**

I. Introduction

NoSQL storage systems are used extensively by online applications and provide an attractive alternative to conventional databases when the need for scalability outweighs the need for transactions. Several of these systems, most notably Cassandra [1], Voldemort and Riak, are derivatives of Amazon's Dynamo [2], and share a common quorum-based replication model that enables different behaviors with respect to consistency. For example, in the context of Brewer's CAP principle, the storage system can be configured to provide either consistency or availability during a network partition [3]. Application developers who use such systems face a choice of multiple client-side "consistency levels" that determine the size of a partial quorum for reads and writes, which is the number of replicas that must respond to a read or write request. This parameter directly affects the latency of read and write operations, and indirectly affects the consistency observed by client applications. For example, strict (e.g., majority) quorums provide so-called "strong consistency," meaning that reads always return the latest value of a data object, whereas non-overlapping partial quorums provide eventual consistency whereby reads may return stale values for some period of time after an update while the replicas of a data object converge to a common state. Terry et al. describe the behavior of eventual consistency more precisely in the context of Bayou: the system "guarantees that all servers *eventually* receive all Writes ... and that two servers holding the same set of Writes will have the *same* data contents" [4].

In the absence of a fine-grained consistency-latency tuning mechanism, application developers are challenged in two ways. First, using conventional client-side consistency settings they are often restricted to extremes, such as eventual versus strong consistency, and have no means to strike a flexible compromise. This is especially problematic when the latencies for strongly and weakly consistent operations differ by orders of magnitude, as in geo-replicated systems, and the latency requirements for a given application lie somewhere in-between. Second, although standard tools and benchmarks (e.g., [5]) can be used to measure latency, tools for quantifying the actual consistency observed by client applications are only starting to emerge from research (e.g., [6]). As a result, an application developer who is concerned about consistency may be compelled to adopt quorum operations, unaware that with a weaker client-side consistency setting the consistency actually observed in practice may be quite strong and well within the application's requirements. For example, this might occur when the workload is light in terms of throughput and is spread out over a large data set, which minimizes the likelihood that a data item is read during the window of inconsistency following an update to that item.

In this paper we investigate the possibility of tuning the consistency-latency trade-off in a more fine-grained manner than is possible using conventional client-side consistency levels. Specifically, we focus on techniques that enable such tuning using a parameter with a continuous range of values, as opposed to selecting consistency levels from a short menu of discrete options (e.g., read one, read majority, read all). Attaining fine-grained control over consistency and latency is an important step on the path to supporting service level agreements (SLAs) that allow applications to request a diverse range of requirements with respect to these quantities. For example, such an SLA might allow a client application to specify either an average latency or a 95th %-ile latency of at most L milliseconds, and a proportion of inconsistent operations (defined more precisely in Section II) of at most $C\%$. In this framework a latency-favoring application such as a shopping cart may specify a lower L and higher C, whereas a consistency-favoring application such as a personal cloud file system may opt for a higher L and lower C. Naturally, such SLAs can also specify guarantees on throughput.

Our main technical contribution in this paper is the empirical evaluation of two techniques for fine-grained consistency-latency tuning. The first, a novel technique we call *continuous partial quorums* (CPQ), entails making a random choice between multiple discrete consistency levels on a per-operation basis. For example, the application may choose consistency level one with probability p and majority quorums with probability $1 - p$. In this case p itself becomes a continuous tunable parameter. In contrast, using fixed consistency levels for reads and writes and a replication factor of three, there are only three possible partial quorums—one, two/quorum, and three/all—and hence only nine discrete combinations. Furthermore, only four of these combinations, namely those using the one and two/quorum consistency levels, provide availability in the presence of a single server failure. The second technique, called *artificial delays* (AD) uses a weak client-side consistency level and boosts consistency by injecting a tunable delay into each storage operation. Intuitively, longer delays allow more time for updates to propagate through the system, which decreases the likelihood of consistency anomalies at the cost of increasing latency. Our experimental evaluation considers both single data center and geo-replicated deployments of Apache Cassandra in Amazon's EC2 environment, and leads to different conclusions in these two cases: CPQ provides better average latencies than AD when nodes are connected using a fast data center network and processing delays dominate operation latencies, whereas AD provides more predictable latencies and a slightly better consistency-latency trade-off when nodes communicate over a high-latency wide area network.

II. Methodology

In this section we describe in detail the techniques, performance metrics, storage system, and benchmark used in our experiments.

A. Techniques

We implement CPQ and AD at clients, and compare them against fixed consistency levels, which is our baseline. CPQ generates for each operation a pseudo-random number and selects between two discrete client-side consistency levels with a tunable probability p. The choice is between consistency level one and majority quorums (2 out of 3 replicas), and is made independently for each operation. AD always uses consistency level one for all operations, and boosts consistency by injecting a tunable artificial delay immediately before a read as well as immediately after a write. For simplicity the same delay is used for all operations, although in practice the delays could be determined independently for reads and writes.

B. Performance Metrics

Our experiments measure two quantities: latency and consistency. Latency is measured easily on a per-operation basis, but consistency is more difficult to quantify because it pertains to the interaction of multiple operations. Our precise notion of consistency is inspired by Lamport's atomic register [7], which in our context represents a key-value pair accessed using read and write operations. Informally speaking, Lamport's atomicity property states that all operations must appear to take effect instantaneously at some point in time between their start and finish. This ensures, for example, that when a write terminates from the point of view of the client, any read started later by any client will obtain either the value assigned by this write or a later value. Atomicity is a property of a trace of operations that records the start and finish times of each operation as well as its arguments and responses, and refers to the consistency actually observed by clients rather than the state of affairs in the back-end implementation of a storage system, where discrepancies among replicas may exist and yet go unnoticed by clients. We believe this is the right paradigm for discussing consistency in the context of SLAs because it separates cleanly the behavior observed by client applications from the storage system's implementation details, which makes it possible to specify application requirements in a system-independent manner.

Lamport's atomicity property alone is not sufficient for our purposes because it can only classify a given trace as either consistent (i.e., atomic) or not, and does not provide a continuous measurable quantity. Instead we adopt the technique of Golab, Li and Shah, which quantifies how far a given trace deviates from atomicity by computing the proportion of operations in a trace that are involved in consistency anomalies [8]. This proportion is zero if the trace is atomic to begin with, and positive otherwise, with higher values indicating more frequent consistency anomalies. In the remainder of the paper we will refer to the metric as the *proportion of inconsistent operations*.

An example of a trace is shown in Figure 1, where each operation is denoted by an interval of time and labeled as follows: $W(v)$ denotes a write of value v and $R(v)$ denotes a read that returns v. All of the operations shown are applied

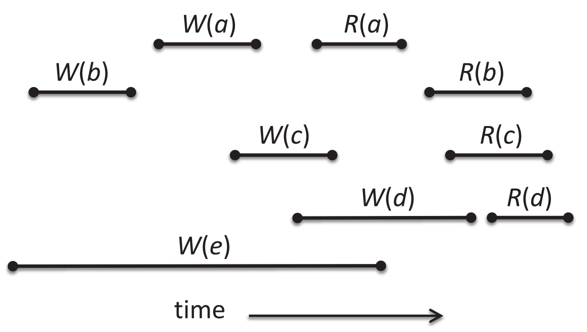


Figure 1. Example trace used for computing the proportion of inconsistent operations.

to the same object, and in general traces for different objects are analyzed separately. The trace is not atomic for several reasons including the following: $W(b)$ must take effect before $W(a)$, which must take effect before $R(b)$. We can view this consistency anomaly as either $R(b)$ returning a stale value, such as when weak client-side consistency settings are used, or the two writes taking effect out of order with respect to their positions in the trace, such as when the storage system internally orders writes using timestamps obtained from loosely synchronized clocks on different hosts. The trace can be made atomic by removing all operations that access value b, namely $W(b)$ and $R(b)$, which allows us to interpret the remaining operations as taking effect in the following order: $W(e), W(a), R(a), W(c), R(c), W(d), R(d)$. Since this transformation removes two out of nine operations in the trace, the proportion of inconsistent operations is calculated as $2/9$, with both $W(b)$ and $R(b)$ viewed as contributing to the consistency anomaly.

The use of larger partial quorums and artificial delays tends to improve (i.e., lower) the value of the proportion metric in the following sense: as the interval from the start time to the finish time of each operation lengthens due to either the additional protocol messages or the artificial delay, the time intervals for pairs of operations tend to overlap and the constraints imposed by Lamport's atomicity property on the order in which operations take effect become less stringent. For example, in Figure 1 extending the time intervals for $W(b)$ and $W(a)$ to the right while extending the interval of $R(b)$ to the left causes all three intervals to overlap eventually, and removes the constraint that $W(a)$ must take effect after $W(b)$ and before $R(b)$. This, in turn, mitigates the consistency anomaly explained earlier.

Computing the proportion of inconsistent operations for an arbitrary trace of operations is an NP-hard problem because it generalizes the NP-complete problem of deciding whether a trace is atomic [9]. However, the proportion can be computed efficiently using a dynamic programming algorithm under the assumption that two writes on the same object always assign distinct values [8].

C. Storage System

Our experimental evaluation uses Apache Cassandra 2.0.10—a popular open-source implementation of a Dynamo-style distributed storage system [1]. The data set is organized internally as a set *keyspaces*, each of which contains a set of *column families* that resemble tables in a conventional database. Replication is configured at keyspace level, and determines the number of server failures that a Cassandra cluster can tolerate without losing data. Clients can execute read and write operations using any node in the cluster, which acts as a coordinator and routes protocol messages to the relevant replicas. The client-side consistency level setting specified by an application determines how many replicas must acknowledge an operation, which affects both the consistency and latency observed by clients. That said, the coordinator in general attempts to access all replicas: for writes it sends updates to all replicas, and for reads it retrieves either the value of a data item, using a *direct request*, or its digest, using a *digest request* to detect discrepancies among replicas. Updates are timestamped, which allows the coordinator to resolve the most recent value of a data item in the event that a discrepancy is detected. The process of detecting and fixing discrepancies in the course of a read operation is called *read repair*.

D. Benchmark

The workload is generated using the Yahoo Cloud Serving Benchmark (YCSB) [5], with a modified Cassandra client connector to support the CPQ and AD techniques. YCSB collects precise measurements of throughput and latency, and the modified client connector records a trace of operations at each host for off-line consistency calculations. Traces from all hosts are merged at the end of each experiment and the result forms the input to the algorithm that computes our chosen consistency metric.

Each experiment comprises a YCSB load phase starting with an empty keyspace, followed by a 180-second YCSB work phase. Each host runs a single YCSB process with 128 client threads that connect to the local Cassandra server. We use a mixture of 80% read and 20% write operations that access 128-byte values. Keys are generated using one of two YCSB probability distributions: "latest" and zipfian, both with a key space of 1000. Read-dominated workloads with skewed distributions are generally representative of online applications, such as social networking, where users spend most of their time reading data, and some data items are much more popular than others. We use small key spaces similarly to [10] because they tend to illicit more frequent consistency anomalies. The replication factor is set to three and a majority quorum is two out of three replicas. Throughput is kept constant using YCSB, as described in more detail in Section III.

III. Experiments

A. Hardware and software environment

The experiments are staged in Amazon's EC2 environment. For simplicity we deploy a small cluster comprising three m3.xlarge on-demand instances, each equipped with four Intel Xeon E5-2670 2.50GHz cores, 16 GB RAM, 2x40 GB SSD local storage. The software environment includes an Ubuntu 14.04 x86_64 image with Linux kernel version 3.13.0 in HVM (Hardware Virtual Machine) mode, Oracle Java 1.7.0_72, Apache Cassandra 2.0.10 and YCSB 0.1.4 modified as explained in Section II. Cassandra is configured with default settings and its internal state (data file, commit log, and cache directories) is placed on one of the SSDs. The second SSD is left unused. Each host runs a single YCSB process with 128 client threads that connect to the Cassandra server on the same host.

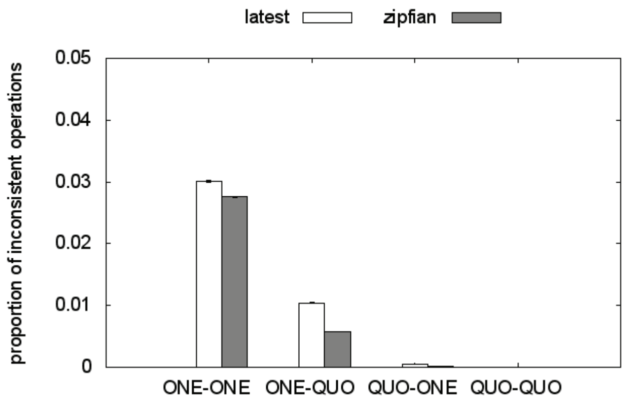
For experiments that use a single data center all three instances are deployed in the us-west-2 region (Oregon), with an RTT generally in the range 150-200 μs. For experiments that use geo-replication we deploy the instances in three different regions: us-west-1 (N. California), us-east-1 (N. Virginia), and sa-east-1 (Sao Paulo, Brazil). The RTT is approximately 35-40ms between the us-west and us-east data centers, and 70-80ms between the us-based regions and sa-east.

Clock synchronization, which is very important for our chosen method of measuring consistency, is achieved using NTP. In the single data center case we set up one host as the NTP server and the other two as clients, which is sufficient to synchronize clocks to within 0.05ms or better. The same approach is not possible in the geo-replicated case due to much higher network latencies and latency variations, and so we use the default external NTP servers. This enables synchronization to within 5-10ms.

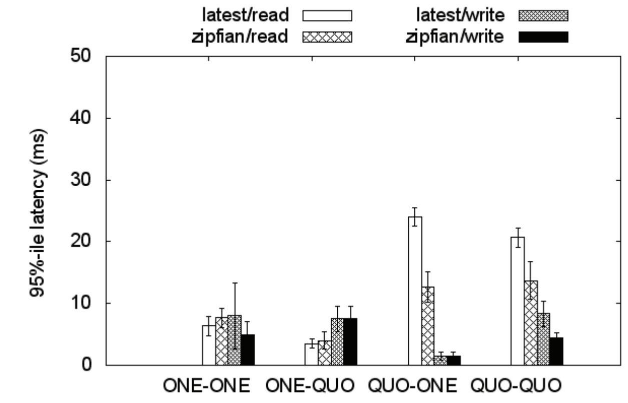
B. Visualizations

We present several types of graphs in this section. Part (a) of Figures 2–7 presents the proportion of inconsistent operations, which was defined in Section II. The error bars shown on the graphs represent one standard error calculated as $\sqrt{\hat{p}(1-\hat{p})/n}$ where $\hat{p}$ is the calculated sample proportion. In general we found that variations in the consistency metric between runs of the same experiment were higher than indicated by the error bars, likely due to small differences in clock synchronization as well as weak performance isolation in EC2. Using single-tenancy instances did not remedy the problem.

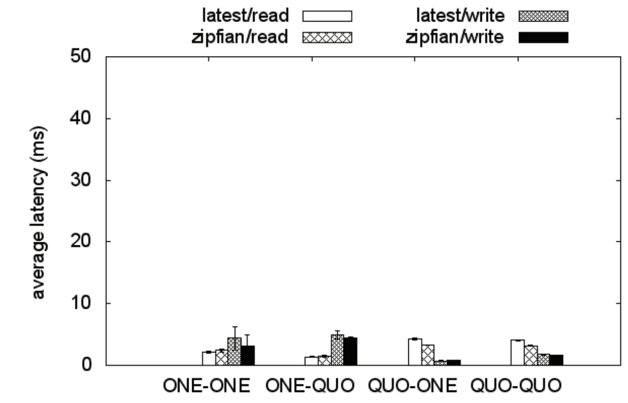
Parts (b) and (c) of Figures 2–7 present the 95%-ile and average latency for each experiment, with separate bars for read and write operations. Error bars on these graphs represent one standard deviation of the sample of three measurements corresponding to the three YCSB clients used, one per host.



(a) consistency vs. consistency level



(b) 95th %-ile latency vs. consistency level

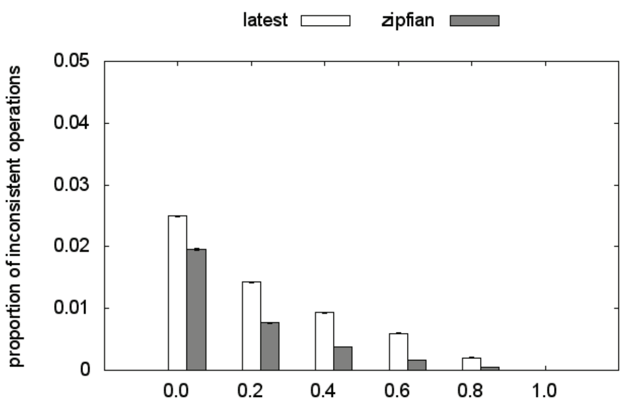


(c) average latency vs. consistency level

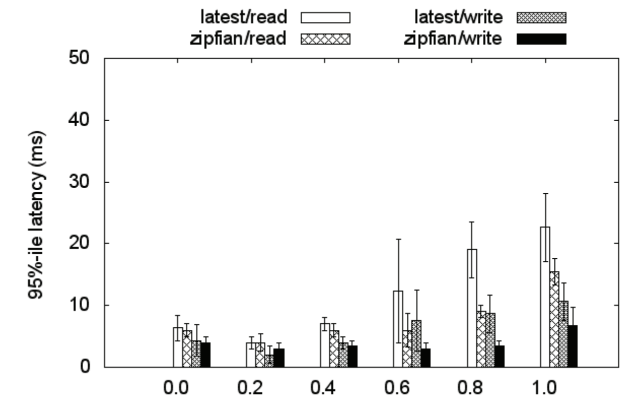
Figure 2. Consistency and latency vs. client-side consistency level (e.g., ONE-QUO means read one, write majority quorum).

C. Single data center results

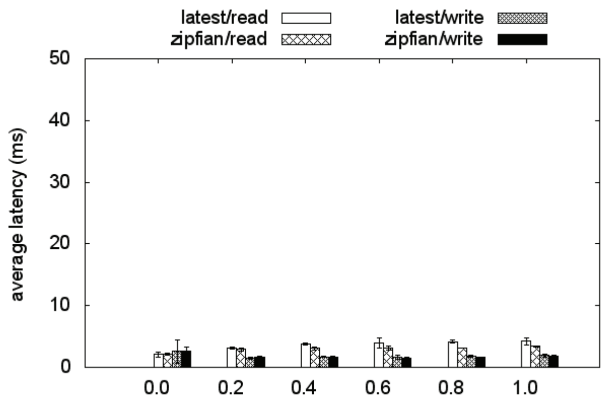
In the first series of experiments, presented in Figures 2–4, all hosts are placed in the same EC2 region and the same availability zone. The maximum throughput for our chosen workloads in this deployment is 3-10kops/s depending on the case. The throughput is limited by the number of YCSB threads in experiments that involve delays, and by other



(a) consistency vs. probability of quorum level

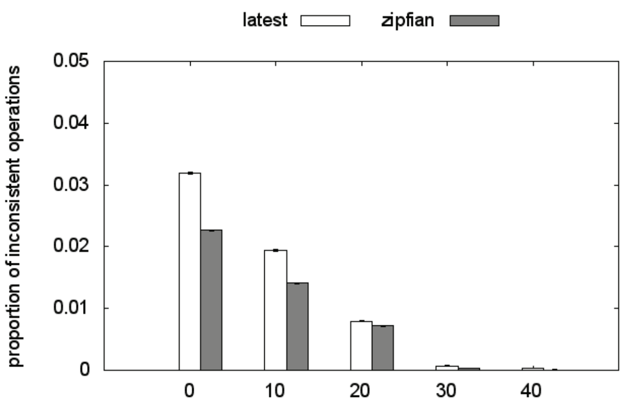


(b) 95th %-ile latency vs. probability of quorum level

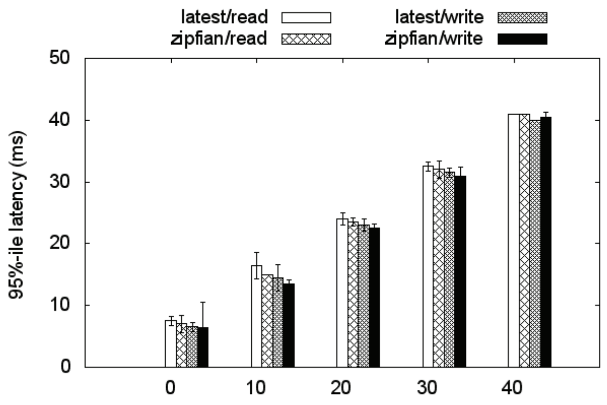


(c) average latency vs. probability of quorum level

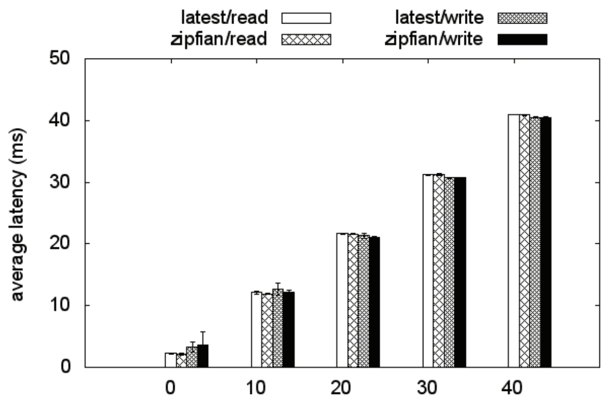
Figure 3. Consistency and latency versus probability of client-side consistency level quorum vs. one.



(a) consistency vs. artificial delay (ms)



(b) 95th %-ile latency vs. artificial delay (ms)



(c) average latency vs. artificial delay (ms)

Figure 4. Consistency and latency versus artificial delay (ms).

factors (CPU and network) in other cases. In the interest of a controlled experiment we fixed the throughput in YCSB using a target of 3kops/s. Actual throughput during experiments is within 1% of the target. The network RTT is less than 1ms and, as shown by Fan et al., consistency is affected substantially by processing delays, particularly the stop-the-world behavior of the Java garbage collector [11]. The stop-the-world pause can delay the processing of updates for up to tens or even hundreds of ms, causing stale reads.

Figure 2 illustrates the consistency-latency envelope of fixed client-side consistency levels, our baseline technique. We focus specifically on different combinations of one and majority quorum consistency levels, which provide availability in the presence of one failed server given the replication factor of three. The x-axis labels are of the form R-W where R and W indicate the client-side consistency levels for reads and writes, respectively. The proportion of inconsistent operations is mostly under 0.03, and is exactly zero for quorum operations, indicating that Cassandra produced an atomic trace. Quorum consistency does not, however, guarantee atomicity in general, as explained in [10].

The latency numbers range from a few ms to near 30ms at the 95th %-ile, and under 10ms for average latency. In many cases the latencies are much higher than the network RTT of 0.15-0.2ms, indicating substantial processing delays. As expected, we observe higher latencies for quorum operations than for operations with consistency level one, especially for reads. Curiously, write latencies are slightly higher when reads with consistency level one are used than with quorum reads. We do not have an explanation for this behavior but we found that it was reproducible.

The second set of results, presented in Figure 3, demonstrates continuous partial quorums in action. In this experiment the client chooses majority quorum consistency with probability p, shown on the x-axis, and one consistency with probability $1 - p$. The same policy is used for both read and writes. As p increases from 0 to 1 we observe that both the consistency and latency gradually morph from values corresponding to the ONE-ONE case in Figure 2 to values corresponding to the QUO-QUO case. Thus, CPQ successfully attains points in the two-dimensional consistency-latency spectrum that lie in-between the discrete points attained using fixed client-side consistency levels. In particular, for $0 < p < 1$ the consistency-latency trade-offs achieve a different balance of read and write latencies than the ONE-QUO and QUO-ONE cases in Figure 2, which are biased toward either reads or writes. Overall CPQ appears most effective at trading off consistency against 95th %-ile latency for reads, which comprise 80% of the workload, with write latencies showing less variation in general.

The last set of results, presented in Figure 4, demonstrate the behavior of artificial delays. The length of the artificial delay in milliseconds is shown on the x-axis, and contributes directly to the latency of read operations. As a result, both 95th %-ile latency and average latency increase gradually from under 10ms to over 40ms as the delay is increased from 0 to 40ms. The consistency, shown in part (a), also varies with the length of the artificial delay, and approaches levels observed with quorum operations in Figure 2 when the delay is 30ms or more. However, in those cases the average latency is about an order of magnitude worse than with quorum operations, and the 95th %-ile latency is also substantially higher. On a positive note, 95th %-ile latencies are slightly more predictable compared to when using fixed consistency levels, and more uniform across probability distributions as well as between reads and writes.

Compared to CPQ we see that artificial delays lead to substantially higher average latencies at the same value of the consistency metric. For example, the average latency for CPQ with probability $p = 0.8$ is under 5ms in Figure 3 (c), whereas an artificial delay of 20ms achieves worse consistency and average latencies of 20-25ms in Figure 4 (c). Similarly, the 95th %-ile latencies in this case are 5-25ms for CPQ and 22-25ms for artificial delays. Overall we conclude that compared to CPQ, artificial delays achieve an inferior trade-off between consistency and latency in the single data center experiments. The predictability of the latencies when artificial delays are used is attractive, but also demonstrates the main weakness of this technique: all operations are delayed uniformly whereas quorum operations in Figures 2 and 3 exhibit latencies that more faithfully reflect the actual processing delays, which vary with time (e.g., garbage collector running versus not) and with the workload (e.g., reads vs. writes).

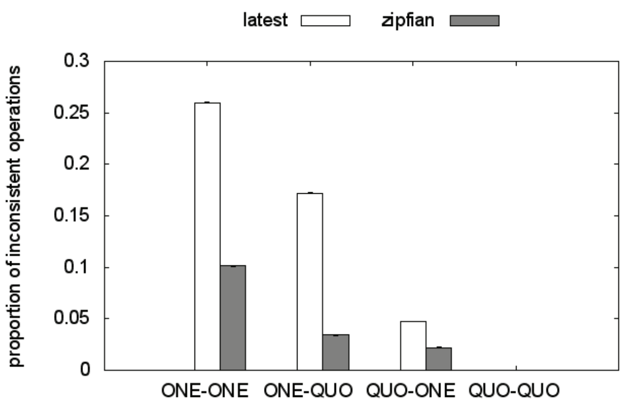
D. Geo-replication results

In the second series of experiments, presented in Figures 5–7, we consider geo-replication. The maximum throughput for our workloads that can be sustained in all experimental cases is 1-2kops/s, and so we fix the target throughput in YCSB at 1kops/s, which the workload generator meets to within 1%. The network RTT is generally in the tens of ms, and so we expect network delays to affect consistency more substantially than in a single data center.

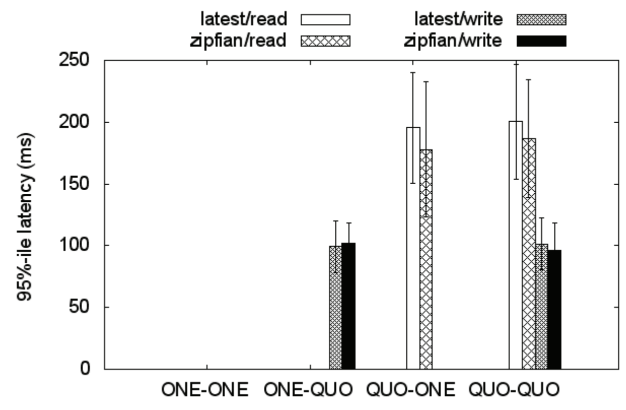
Figure 5 shows a pattern similar to Figure 2, but with sharper differences in latency between different combinations of client-side consistency settings. The proportion of inconsistent operations is also generally higher than in the single data center case, exceeding 0.25 in the ONE-ONE case. This is expected since updates take much longer to propagate from the coordinator to the replicas.

Figure 6 also shows some similarity to Figure 3. In parts (a) and (c) the numbers gradually transition from levels corresponding to fixed ONE-ONE consistency to levels corresponding to QUO-QUO consistency. Part (b), on the other hand, shows an interesting contrast. Whereas in Figure 3 the 95th %-ile latency does not show much variation, in Figure 6 there is an abrupt increase only from $p = 0$ to $p = 0.2$, and after that the values stabilize. In general as soon as p exceeds 0.05, the 95th %-ile latency is dominated by the subset of operations that use quorum consistency. However, in this particular case the difference in latency between consistency level one and majority quorums is much higher than in Figure 3, where processing delays (e.g., garbage collection and compaction) appear to dominate. Thus, CPQ successfully attains a continuous range of points in the two-dimensional consistency-latency spectrum but only with respect to average latency.

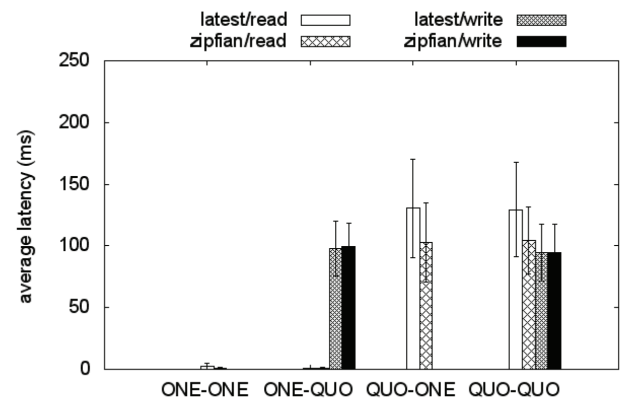
Artificial delays work much more effectively in the geo-replicated case than in a single data center, as shown in Figure 7. The delay, which now ranges from 0 to 100ms to compensate for the larger RTT, has the desired effect on the



(a) consistency vs. consistency level

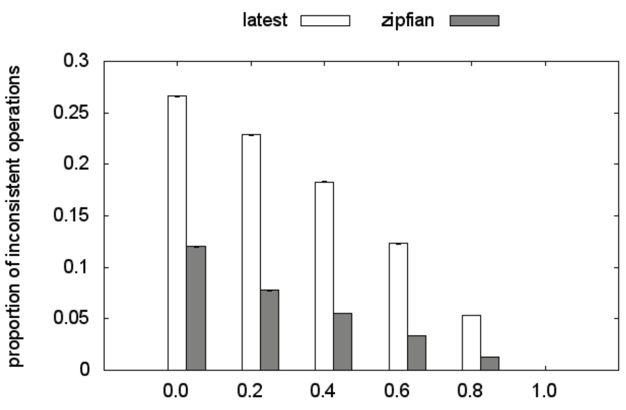


(b) 95th %-ile latency vs. consistency level

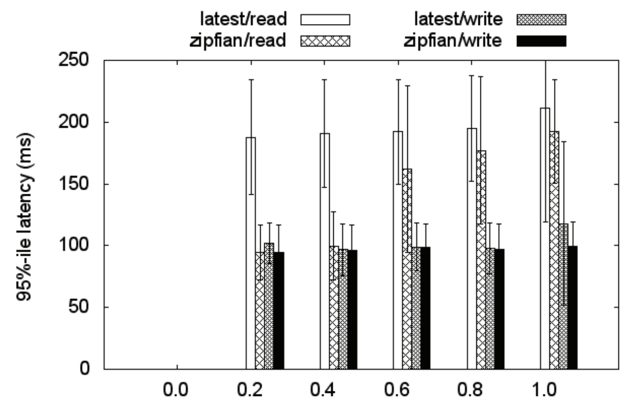


(c) average latency vs. consistency level

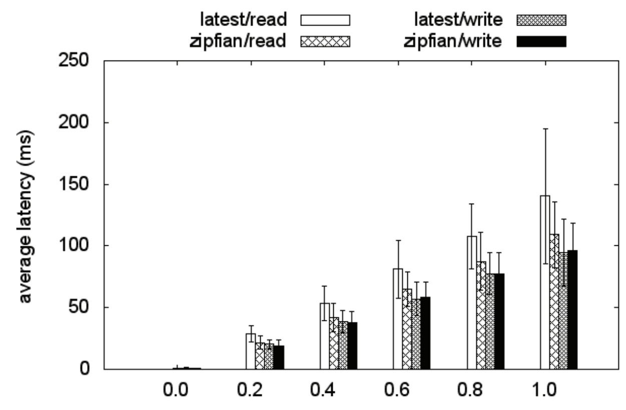
Figure 5. Consistency and latency vs. client-side consistency level (e.g., ONE-QUO means read one, write majority quorum).



(a) consistency vs. probability of quorum level



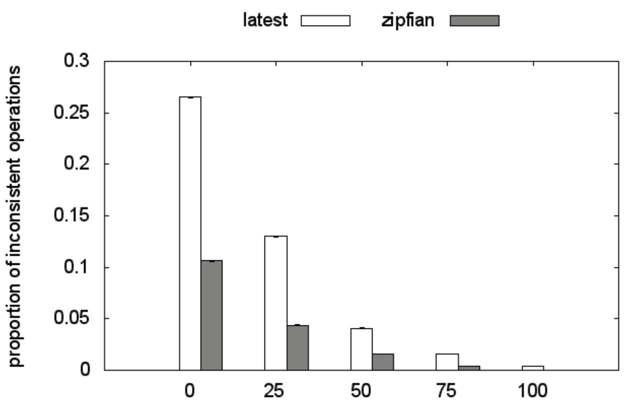
(b) 95th %-ile latency vs. probability of quorum level



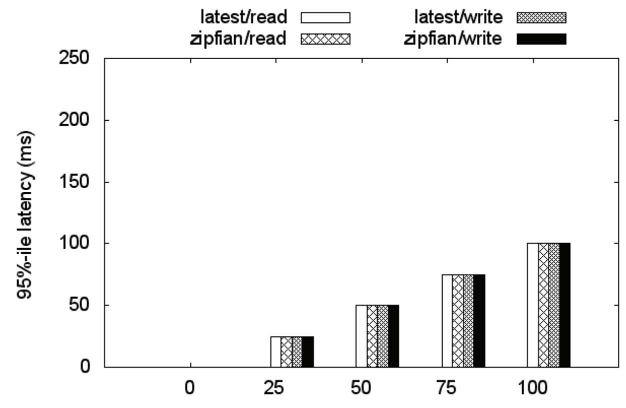
(c) average latency vs. probability of quorum level

Figure 6. Consistency and latency versus probability of client-side consistency level quorum vs. one.

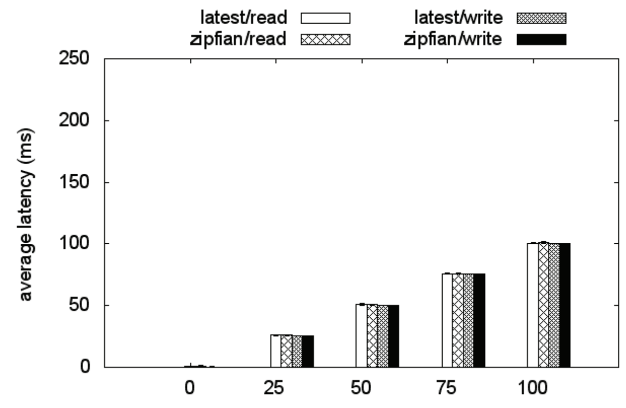
consistency metric, although once again it is not necessarily able to remove all consistency anomalies at its maximum value. The latency also varies gradually, increasing from a few ms to approximately 100ms, and generally corresponds to the latency of a ONE-ONE operation plus the length of the artificial delay. This holds for both average and 95th %-ile latency, in contrast to CPQ in Figure 6 (b). In fact, the latencies with a 100ms delay are somewhat better than using fixed consistency level QUO-QUO in Figure 5, especially at the 95th %-ile, but with only slightly worse consistency. At 50ms the consistency with artificial delays is better than for CPQ with $p = 0.8$, and the latency is much better, once again especially at the 95th %-ile. Thus, artificial delays demonstrate measurable advantages over CPQ.



(a) consistency vs. artificial delay (ms)



(b) 95th %-ile latency vs. artificial delay (ms)



(c) average latency vs. artificial delay (ms)

Figure 7. Consistency and latency versus artificial delay (ms).

IV. Related Work

Recent research in the area of consistency has addressed the classification of consistency models, consistency measurement, and the design of storage systems that provide precise consistency guarantees. This body of work is influenced profoundly by the CAP principle, which states that a distributed storage system must make a trade-off between consistency (C) and availability (A) in the presence of a network partition (P) [3]. The PACELC formulation builds on CAP by considering two separate cases: during a network partition it reduces directly to CAP, but during failure-free operation it dictates a trade-off between latency and consistency [12].

Distributed storage systems use a variety of designs that achieve different trade-offs with respect to CAP. Amazon's Dynamo and its derivatives (Cassandra, Voldemort and Riak) use a quorum-based replication scheme that can operate either in CP (i.e., strongly consistent but sacrificing availability) or AP (i.e., highly available but eventually consistent) mode depending on the size of the partial quorum used to execute read and writes [1], [2]. The techniques discussed in this paper–CPQ and AD—are targeted specifically at this family of systems. Since they are implemented at clients these techniques can be used with any quorum-replicated system that supports tunable partial quorums.

Many alternative designs have been proposed for supporting stronger notions of consistency in storage systems. Bigtable provides atomic access to individual rows, and is eventually consistent when deployed across multiple data centers [13]. PNUTS provides per-record timeline consistency, which ensures that replicas of a record apply updates in the same order [14]. COPS provides causal consistency with convergent conflict handling and read-only transactions, and is designed for wide-area deployments [15]. Causal consistency is in some sense the strongest property that can be guaranteed in the presence of network partitions, which makes COPS an AP system in the context of CAP [16]. Bolt-on causal consistency is a shim layer that provides causal consistency on top of eventual consistency [17]. Spanner is a geo-replicated transactional database that provides external consistency, which is similar in spirit to Lamport's atomicity property (see Section II) [18]. The replication and transaction commitment protocols in these systems are geared toward specific notions of stronger-than-eventual consistency and do not expose a client-side consistency level setting that could be used with our CPQ technique.

Several systems consider the problem of providing continuously tunable consistency guarantees. TACT is a middleware layer that uses three metrics to express consistency requirements with respect to read and write operations: numerical error, order error, and staleness [19]. TACT relies on a consistency manager that pushes updates synchronously to other replicas. Pileus allows client applications to declare consistency and latency requirements in the form of SLAs [20]. These SLAs include latency and staleness bounds but do not support the types of probabilistic guarantees discussed in Section I. Internally, Pileus enforces the SLAs by choosing which replica to access in an SLA-aware manner, whereas Dynamo-style systems tend to always access the closest replicas. Tuba supports consistency SLAs by automatically reconfiguring the locations of its replicas in response to the client's location and request rates [21]. AQuA is middleware layer that allows the client application to specify latency and consistency requirements similarly to Pileus, but with a focus on time-sensitive applications [22]. It provides probabilistic timeliness guarantees by selecting replicas dynamically using probabilistic models.

We are aware of only two systems that use artificial delays for consistency-latency tuning. Golab and Wylie propose consistency amplification—a framework for supporting consistency-based SLAs by injecting client-side or server-side delays whose duration is determined adaptively using measurements of the consistency actually observed by clients [23]. Rahman et al. present a similar system called PCAP, where delays are injected only at clients and their duration is determined using a feedback control mechanism [24]. PCAP also varies the read repair rate, which is shown to be a far less effective tuning knob. The evaluation of the system considers the proportion of operations that satisfy particular consistency and latency requirements, and does not investigate the optimality of this trade-off with respect to fixed client-side consistency levels such as majority quorums. The argument given against strict quorums is that they may cause storage operations to block in the event of a network partition. However, the consistency calculations used to tune artificial delays in PCAP are themselves blocking because they are based upon operation logs collected from multiple servers. Furthermore, in practice even quorum operations can be made non-blocking by using read and write timeouts, which are configurable in recent versions of Cassandra. Timeouts ensure that every operation eventually either completes successfully, or fails and allows the client to retry the operation using a smaller partial quorum.

The use of server-side artificial delays is explored in [11] as a technique for reducing the severity of consistency anomalies in Cassandra when client-side consistency level ONE is used. The delays are injected judiciously following the garbage collection stop-the-world pause, which improves consistency drastically with negligible impact on latency. In contrast, the artificial delays used in PCAP and explored in our own experiments incur a latency penalty for every single read operation, which increases average latency directly.

In the pursuit of an empirical understanding of CAP-related trade-offs several papers have explored techniques for measuring consistency [10], [25], [26], [27]. Measuring consistency in a precise way is subtly difficult because consistency anomalies such as stale reads are the result of interplay between multiple storage operations. As a result, some of the contributions in this space consider simplified techniques that measure the convergence time of the replication protocol rather than the consistency actually observed by client applications (e.g., [25], [26]) or quantify the consistency observed in terms of quantities that do not translate directly into staleness measures expressed naturally in units of time (e.g., counting cycles in a dependency graph [27]). Probabilistically bounded staleness (PBS) is a mathematical model of partial quorums that overcomes these limitations but is based upon the simplifying assumption that writes do not execute concurrently with other operations [6]. The theory underlying probabilistic quorum systems was originally developed by Malkhi, Reiter, and Wright [28].

V. Discussion and Conclusion

Our experiments using Cassandra in Amazon's EC2 environment show that the consistency-latency trade-off can be tuned in a continuous manner using mechanisms that operate on top of a handful of discrete client-side consistency levels. We demonstrate this point using a novel technique called continuous partial quorums (CPQ), which chooses randomly between two discrete consistency levels according to a tunable probability parameter, as well as the known technique of injecting artificial delays (AD). In a single data center environment CPQ is able to effectively bridge the gap between a pair of fixed client-side consistency levels, and generally achieves a more attractive consistency-latency trade-off than AD, which delays all operations uniformly without regard to variations in processing delays. This aspect of our results confirms informal claims regarding the potentially detrimental effect of injecting artificial delays (e.g., see [6]), albeit only in the single data center case. On the other hand in a geo-replicated environment AD produces latencies that are both more predictable and slightly lower than CPQ for the same degree of consistency. That said, AD cannot guarantee strong consistency unless the length of the delay can be guaranteed to exceed both network and processing delays.

Although we evaluate CPQ and AD specifically in the context of Apache Cassandra, both techniques are applicable to any system that supports a set of discrete client-side consistency options. In future work we plan to implement and evaluate these techniques on top of other storage systems and develop techniques for automating the choice of values for their corresponding tuning knobs.

VI. Acknowledgment

We are grateful to the anonymous referees for their helpful comments and suggestions. The authors are supported by funding from Hewlett-Packard Labs, Google, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

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

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

[3] E. A. Brewer, “Towards robust distributed systems (Invited Talk),” in *Proc. of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2000.

[4] 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.

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

[6] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, “Probabilistically bounded staleness for practical partial quorums,” *PVLDB*, vol. 5, no. 8, pp. 776–787, 2012.

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

[8] W. Golab, X. Li, and M. A. Shah, “Analyzing consistency properties for fun and profit,” in *Proc. of the 30th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2011, pp. 197–206.

[9] P. Gibbons and E. Korach, “Testing shared memories,” *SIAM Journal on Computing*, vol. 26, pp. 1208–1244, 1997.

[10] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta, “Client-centric benchmarking of eventual consistency for cloud storage systems,” in *Proc. of the 34th International Conference on Distributed Computing Systems (ICDCS)*, 2014, pp. 493–502.

[11] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, and B. Wong, “Understanding the causes of consistency anomalies in Apache Cassandra,” *PVLDB*, vol. 8, no. 7, pp. 810–813, 2015.

[12] 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.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “Pnuts: Yahoo!’s hosted data serving platform,” *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[15] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS,” in *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 401–416.

[16] P. Mahajan, L. Alvisi, and M. Dahlin, “Consistency, availability, and convergence,” *University of Texas at Austin Tech Report*, vol. 11, 2011.

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

[18] 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.

[19] 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, 2002.

[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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 309–324.

[21] M. S. Ardekani and D. B. Terry, “A self-configurable geo-replicated cloud storage system,” in *Symp. on Op. Sys. Design and Implementation (OSDI)*, 2014, pp. 367–381.

[22] S. Krishnamurthy, W. H. Sanders, and M. Cukier, “An adaptive quality of service aware middleware for replicated services,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 1112–1125, 2003.

[23] W. Golab and J. J. Wylie, “Providing a measure representing an instantaneous data consistency level,” US Patent Application 20,140,032,504, filed 2012, published 2014.

[24] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya, “Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores,” 2015, http://arxiv.org/abs/1509.02464.

[25] 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.

[26] 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.

[27] K. Zellag and B. Kemme, “How consistent is your cloud application?” in *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC)*, 2012, p. 6.

[28] D. Malkhi, M. Reiter, and R. Wright, “Probabilistic quorum systems,” in *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1997, pp. 267–273.