Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services

David Bermbach[1], Liang Zhao[2], and Sherif Sakr[2]

[1] Karlsruhe Institute of Technology
Karlsruhe, Germany
david.bermbach@kit.edu
[2] NICTA and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract. The CAP theorem and the PACELC model have described the existence of direct trade-offs between consistency and availability as well as consistency and latency in distributed systems. Cloud storage services and NoSQL systems, both optimized for the web with high availability and low latency requirements, hence, typically opt to relax consistency guarantees. In particular, these systems usually offer eventual consistency which guarantees that all replicas will, in the absence of failures and further updates, eventually converge towards a consistent state where all replicas are identical. This, obviously, is a very imprecise description of actual guarantees.
Motivated by the popularity of eventually consistent storage systems, we take the position that a standard consistency benchmark is of great practical value. This paper is intended as a call for action; its goal is to motivate further research on building a standard comprehensive benchmark for quantifying the consistency guarantees of eventually consistent storage systems. We discuss the main challenges and requirements of such a benchmark, and present first steps towards a comprehensive consistency benchmark for cloud-hosted data storage systems. We evaluate our approach using experiments on both Cassandra and MongoDB.

1 Introduction

Recently, we have been witnessing an increasing adoption of cloud computing technologies in the IT industry. This new trend has created new needs for designing cloud-specific benchmarks that provide the ability to conduct comprehensive and powerful assessments for the performance characteristics of cloud-based systems and technologies [8, 15]. These benchmarks need to play an effective role in empowering cloud users to make better decisions regarding the selection of adequate systems and technologies that suit their application's requirements. In general, designing a good benchmark is a challenging task due to the many aspects that should be considered and which can influence the adoption and the

usage scenarios of the benchmark. In particular, a benchmark is considered to be good if it can provide true and meaningful results for all of its stakeholders [17].

Over the past decade, rapidly growing Internet-based services such as e-mail, blogging, social networking, search and e-commerce have substantially redefined the way consumers communicate, access contents, share information and purchase products. Relational database management systems (RDBMS) have been considered as the *one-size-fits-all* solution for data persistence and retrieval for decades. However, the ever increasing need for scalability and new application requirements have created new challenges for traditional RDBMS. Recently, a new generation of low-cost, high-performance database systems, aptly named as NoSQL (Not Only SQL), has emerged to challenge the dominance of RDBMS. The main features of these systems include: ability to scale horizontally while guaranteeing low latency and high availability, flexible schemas and data models, and simple low-level query interfaces instead of rich query languages [22].

In general, the CAP theorem [10] and the PACELC model [1] describe the existence of direct tradeoffs between consistency and availability as well as consistency and latency. These trade-offs are a continuum, so that, due to the popularity of NoSQL systems, there is now a plethora of storage systems covering a broad range of consistency guaranteess. In practice, most cloud storage services and NoSQL systems (e.g., Amazon SimpleDB[3], Amazon Dynamo [14], Google BigTable [11], Cassandra [19], HBase[4]) opt for low latency and high availability and, hence, apply a relaxed consistency policy called *eventual consistency* [25] which guarantees that all replicas will, in the absence of failures and further updates, eventually converge towards a consistent state where all replicas are identical. For situations without failures, the maximum size of the inconsistency window can be bounded based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme, e.g., see [3]. In practice, the implementation and the performance of the eventually consistent mechanism could vary between systems depending on several factors such as the data replication and synchronization protocols, the system load etc. For example, the results of [5] cannot be entirely explained by the aforementioned influence factors.

Motivated by the increasing popularity of eventually consistent cloud-hosted data storage systems, we take the position that a standard consistency measurement benchmark for cloud-hosted data storage system is of great practical value. For example, in cloud environments, users often want to monitor the performance of their services in order to ensure that they meet their Service Level Agreements (SLAs). Therefore, if consistency guarantees are specified as part of the SLA of a cloud-hosted data storage service and the severity of SLA violations can be detected and quantified in an agreeable way, then users could at least receive some monetary compensation.

Furthermore, we believe that a comprehensive consistency benchmark is necessary to evaluate the emerging flow of eventually consistent storage systems.

[3] http://aws.amazon.com/simpledb/
[4] http://hbase.apache.org/

Such a consistency benchmark should be able to provide a clear picture of the relationship between the performance of the system under consideration, the benchmarking workloads, the pattern of failures and the different consistency metrics that can be measured from both of the system perspective and the client perspective. This paper is intended as a call for action; its goal is to motivate further research on building a standard benchmark for quantifying consistency guarantees and behavior of cloud-hosted data storage systems. In this paper, we do not present a comprehensive benchmark that would address all the challenges such a benchmark would need to consider. We do, however, define the main requirements for designing and building this benchmark and present the first steps towards a comprehensive consistency benchmark. In particular, we summarize the main contributions of this paper as follows:

- The identification of the challenges that a comprehensive consistency benchmark should consider.
- An analysis of state-of-the-art consistency benchmarking of NoSQL systems.
- The extension of an existing benchmarking approach towards meeting the defined consistency measurement challenges.
- An experimental evaluation of two popular NoSQL systems, Cassandra [19] and MongoDB[5].

The reminder of this paper is organized as follows. We start with some background on consistency perspectives as well as consistency metrics and identify challenges of a comprehensive consistency benchmark in section 2. Next, in section 3 we describe the extensible architecture of a consistency benchmarking system and its implementation. Afterwards, we use our proposed system for evaluating and analyzing the effects of geo-replication under different workloads on the performance of consistency guarantees for Cassandra and MongoDB in section 4. Section 5 summarizes the related work before we conclude the paper in Section 6.

2 Consistency Measurement: Perspectives, Metrics and Challenges

2.1 Consistency Perspectives

There are two main perspectives on consistency measurement: the perspective of the *provider* of a storage system and the perspective of a *client* of such a system. The provider perspective focuses on the internal synchronization processes and the communication between replicas and is, hence, called *data-centric* consistency. In contrast, the client perspective focuses on the consistency guarantees that a client will be able to observe. This perspective is called *client-centric* consistency [24]. Depending on the perspective, different aspects need to be measured. Figure 1 shows a schematic overview of both consistency perspectives.

For both perspectives, there are two dimensions: *staleness* and *ordering*. Staleness describes how much a replica (or a datum returned to the client) lags behind in comparison to the newest version. It can be expressed both in terms of

[5] mongodb.org

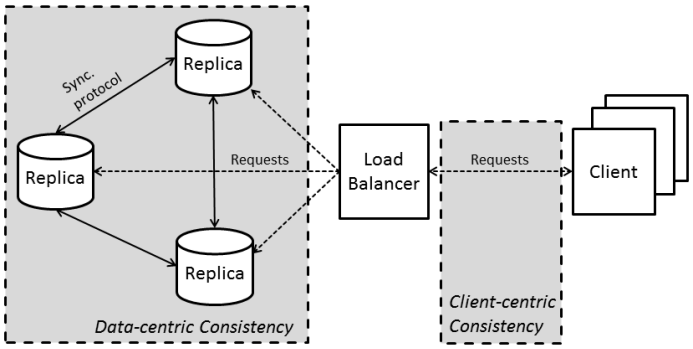


Fig. 1: Data-centric and Client-centric Consistency Perspectives

time or versions and most real world applications can tolerate small staleness values. Data-centric staleness is an upper bound for client-centric staleness [5].

Ordering on the other hand is more critical. It describes how updates are executed on replicas for data-centric consistency and what kind of operation ordering becomes visible to clients for client-centric consistency models. Typical data-centric models like *Sequential Consistency* or *Causal Consistency* [24] do not consider staleness and can be ordered by their strictness. Typical client-centric ordering models like monotonic reads or read your writes are disjunct in their guarantees [6, 24].

2.2 Metrics

From the data-centric consistency perspective, consistency metrics (e.g., staleness or violations of certain ordering models) can be easily determined by analyzing detailed logs created by the different replicas. It is therefore not possible to quantify data-centric consistency for hosted cloud storage services (e.g., Amazon S3) as access to the machines running the actual replicas is required. On the other hand, analyzing those logs after running a standard database benchmark is relatively straightforward. Based on previous work [3, 5, 6], we propose to use fine-grained client-centric metrics. These are useful to an application developer as they provide direct insight into the guarantees an application will encounter and can be measured with any kind of storage system which is treated as a black box. This is also a common practice for measuring the performance benchmarks of database[6] and NoSQL systems [12, 20]. Furthermore, using such an approach does not preclude the usage of replica logs (if available) to also determine data-centric consistency guarantees.

In principle, we suggest that a comprehensive consistency benchmark should include the following staleness metrics:

– *Time-Based Staleness (t-visibility)*: This metric describes how stale a read is in terms of time. The inconsistency window can be calculated as the time

[6] tpc.org

window in between the latest possible read of version n and the start of the write of version $n+1$. Several measurements can be aggregated into a density function describing the distribution of inconsistency windows. If a sufficient number of reads was executed during the inconsistency window, it is also possible to report a cumulative density function describing the likelihood of fresh and stale reads as a function of the duration since the last update.
- *Operation Count-Based Staleness (k-staleness)*: This metric is based on the number of intervening writes and measures the degree of staleness. It obviously depends on the write load on the system and can, thus, be expressed as a function of the write load combined with t-visibility.

Regarding the ordering dimension, the following four client-centric consistency models have been proposed, e.g., see [24, 25]:
- *Monotonic Read Consistency (MRC)*: After reading a version n, a client will never again read an older version.
- *Monotonic Write Consistency (MWC)*: Two writes by the same client will (eventually but always) be serialized in chronological order.
- *Read Your Writes Consistency (RYWC)*: After writing a version n, the same client will never again read an older version.
- *Write Follows Read Consistency (WFRC)*: After reading a version n, an update by the same client will only execute on replicas that are at least as new as version n.

For these four models, we propose to use the likelihood of a violation as metrics. WFRC is usually not directly visible to a client and is therefore hard to determine without access to the replica servers' logs. A standard benchmark, hence, need only include measurements of MRC, RYWC and MWC.

2.3 Measurement Challenges

Accuracy and Meaningfulness of Measurements In general, fine-grained metrics are better for controlling the quality of a system than coarse-grained metrics as they allow the definition of more expressive tradeoff decisions between conflicting design decisions. In practice, an accurate measurement for consistency metrics is a challenging process. For example, the accuracy of client-centric t-visibility measurements is directly influenced by the precision of the clock synchronization protocol. There are several synchronization protocols that work for different scenarios. For example, NTP[7] which is frequently used in distributed systems offers about single digit millisecond accuracy

In addition, apart from workloads which may run in parallel to the benchmark and, thus, use different system resources up to saturation levels, there is also the workload (or rather the interaction pattern between benchmarking clients and datastore) of the benchmark itself. Our experience has shown that observed consistency ordering guarantees are highly volatile in regards to small changes in this workload pattern (e.g., see [7]). Also, there is much interdependency between the actual storage system, the load balancer used and the application

[7] ntp.org

implementation. All in all, this leads to a situation where it is very hard to precisely reproduce a concrete workload on one storage system, not to mention on more than one, in a comparable way.

In large numbers of experiments, we have seen that more simplistic workloads are easier to reproduce and, thus, allow a fairer comparison of systems. At the same time, such a workload is not necessarily representative of an actual application. Storage systems with at least causally consistent guarantees will assert those guarantees independent of the actual workload. For eventually consistent systems, though, some systems might (depending on the load balancer strategy as well as the actual workload) behave like a strictly consistent database in one scenario and become completely inconsistent in another. To us, the best strategy for measuring the ordering dimension is still an unsolved challenge. We believe, though, that reproducible and comparable results are paramount to benchmarking whereas application-specific measurements belong in the area of consistency monitoring. Hence, we tend to favor more simplistic workloads.

Staleness, on the other hand, can be measured independent of benchmarking workloads. Finally, measurement results should be meaningful to application developers in that measured values have a direct impact on application design.

Workloads Modern web-based application are often periodically demanding (e.g. on specific day, month or time of the year) or create bursty workloads that may grow very rapidly before shrinking back to previous normal levels [9]. Ideally, a cloud-hosted data storage service should be infinitely scalable and instantaneously elastic and, thus, be able to handle such a load variance. In particular, a perfectly elastic cloud-hosted storage system should scale up or out its resources indefinitely with increasing workload, and this should happen instantly as the workload increases with no degradation on the application performance.

However, reality is not perfect: In practice, systems use different mechanisms to scale horizontally. For example, when new nodes are added to the cluster, Cassandra moves data stored on the old nodes to the new nodes that have just been bootstrapped. HBase, in contrast, acts as a cache for the data stored in the underlying distributed file system and pulls data only on cache misses. Clearly, reducing the amount of the data that needs to be moved during the bootstrapping process asserts that the system will reach its stable state faster with less congestion on system resources. In other scenarios, live migration techniques are used in a multi-tenancy environment to migrate the tenant with excessive workload to less loaded server in order to cope with increasing workload.

These different implementations for the different systems could affect the consistency guarantees in different ways during the scaling process and should, hence, be considered within a comprehensive assessment of a storage system's consistency guarantees. Previous studies did not consider different workloads (e.g., sinusoidal workloads, exponentially bursty workloads, linearly increasing workload, random workload) and how the system's process of coping with it affects consistency guarantees.

Geo-replication In general, Cloud computing is a model for building scalable and highly available low latency services on top of an elastic pool of configurable virtualized resources such as virtual machines, storage services and virtualized networks. These resources can be located in multiple data centers that are geographically located in different places around the world which provides the ability to build an affordable geo-scalable cloud-hosted data storage service that can cope with volatile workloads. In practice, most of the commercial cloud storage services such as Amazon S3 or SimpleDB do not use wide area replication (only within a region). However, other systems such as PNUTS [23], Megastore [4] and Spanner [13] have been specifically designed for geo-replicated deployments. Using compute services, it is easily possible to deploy geo-replicated NoSQL systems of any kind.

Zhao et al. [28, 29] have conducted an experimental evaluation of the performance characteristics of asynchronous database replication of database servers which are hosted in virtual machines using wide area replication. The results of the study show that an increased application workload directly affects the update propagation time. However, as the number of database replicas increases, the replication delay decreases. Obviously, the replication delay is more affected by the workload increase than the configurations of the geographic location of the database replicas. So far, there is no study that has considered measuring the consistency guarantees of cloud-hosted data storage services in geo-replicated deployments. This issue should be considered in a comprehensive consistency benchmark. Specifically, such a benchmark should analyze the impact of different levels of geo-distribution on consistency guarantees.

Multi-tenancy Multi-tenancy is an optimization mechanism for cloud-hosted services in which multiple customers are consolidated onto the same system so that the economy of scale principles help to effectively drive down the operational cost. One challenge of multi-tenancy in cloud storage services is to achieve complete resource separation and performance isolation of tenants hosted on the same physical server. In practice, the performance for any hosted tenant can turn to be a function of the workloads of other tenants hosted on the same server. A comprehensive benchmark should consider all kinds of cross-effects that could happen between the different tenants.

Node Failure Consideration Inconsistencies in cloud storage systems are often caused by failures. While it is certainly interesting to consider failures, this is not possible when running black box tests, e.g., against cloud storage services, where injecting artificial failures is not an option. If access to the replica servers is possible, a comprehensive benchmark should also consider the effects of different failure types (e.g., node crash-stop, crash-recover or byzantine) on the consistency guarantees of the underlying storage system.

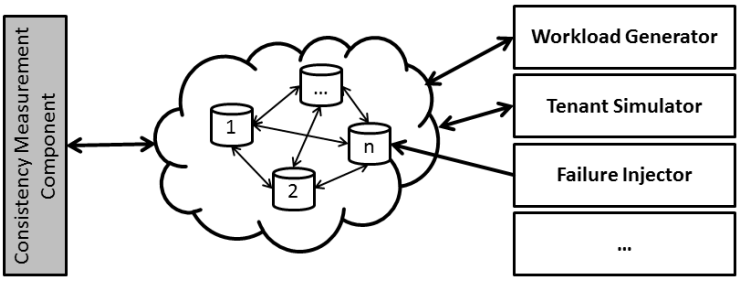


Fig. 2: Benchmark Architecture

3 Consistency Benchmark Design

3.1 Benchmark Architecture

A comprehensive consistency benchmark needs to consider the challenges pointed out in the previous section. From a more technical perspective, it is desirable to reuse existing components and to assert that the benchmark is extensible and flexible. We propose to use a plugin model where the component which is actually measuring consistency is augmented with additional modules if desired. Figure 2 illustrates the basic architecture of our framework with the following main components:

- **Workload Generator**: This component is used to create different workloads on the system to allow the quantification of consistency effects during phases of resource saturation. It should also report results for standard performance metrics like latency or throughput to quantify tradeoffs between consistency and performance.
- **Tenant Simulator**: The Tenant Simulator is used to create a specific kind of behavior for individually simulated tenants of a storage system. While the workload generator just creates load on the system, this component might create a more detailed behavior of a single tenant so that multi-tenant cross-effects on consistency can be studied.
- **Consistency Measurement Component (CMC)** This is the component which is responsible for measuring the consistency guarantees of the underlying system. Its output should use meaningful and fine-grained consistency metrics from a client perspective.
- **Failure Injector**: The Failure Injector is a component which can be used with self-hosted storage systems and can cause a variety of failures.

It could also be reasonable to include a benchmark scheduling and deployment component, e.g., [18], to ease benchmarking of various configurations and systems.

3.2 Benchmark Implementation

For the implementation, we propose to reuse existing, proven tools and to patch them together using shell scripts. The consistency benchmarking tool of Bermbach

and Tai [5] has been used for a large number of consistency benchmarks with various storage systems and services. We extended it slightly to also measure violations of RYWC and MWC so that it, combined with the existing code, measures data for all metrics discussed above. As these continuous, and thus fine-grained, consistency metrics take a client perspective they should be meaningful to application developers. As the benchmarking approach itself relies on a distributed deployment it lends itself to studying the effects of geo-replication. An extension, measuring consistency after delete operations, is currently being developed. Therefore, we will use this tool as our CMC.

The Yahoo! Cloud Servicing Benchmark (YCSB) [12] is the most well known benchmarking framework for NoSQL databases. The tool supports different NoSQL databases and various kinds of workloads and has been designed to be extensible in both dimensions. We will use it as our Workload Generator Component.

So far, we have not included implementations for a Tenant Simulator which is ongoing work at KIT. We have also not used a Failure Injector but Simian Army[8], which was published as open source by Netflix[9], is a promising candidate for future experiments.

The benchmarking tool is extensible for use with all kinds of storage systems. Both our CMC as well as YCSB use an adapter model where the tool itself interacts only with an abstract interface while concrete implementations describe the mapping to the storage system itself. The CMC requires only a key-value interface (even though more complex interfaces can be studied as well) which can be fulfilled by all kinds of systems. YCSB uses the abstract operations insert, update, delete, read and scan for different workloads. Depending on the system itself and the kind of workloads whose influence shall be studied, different combinations of those operations can be used. A Failure Injector could also use a multi-cloud library to create machine failures as well as a similar database adapter framework to cause database failures. The Tenant Simulator could use the same adapter framework as YCSB.

4 Evaluation

To show the applicability of our consistency benchmarking approach, we studied how geo-distribution of replicas combined with two different workloads affects the consistency guarantees of Cassandra and MongoDB. We chose these systems as Cassandra is a popular example of a peer-to-peer system whereas MongoDB is typically (and was during our tests) configured in a master slave setup.

4.1 Experiment Setup

For our evaluation, we ran the following three benchmarks on Amazon EC2[10], each with Cassandra and MongoDB:

[8] github.com/Netflix/SimianArmy
[9] netflix.com
[10] aws.amazon.com/ec2

- **Single-AZ**: All replicas were deployed in the region *eu-west* within the same availability zone[11].
- **Multi-AZ**: One replica is deployed in each of the three availability zones of the region *eu-west*.
- **Multi-Region**: One replica is deployed in three different regions: *eu-west*, *us-west* (northern California) and *asia-pacific* (Singapore).

All replicas were deployed on *m1.medium* instances, whereas the CMC was running on *m1.small* instances distributed according to the respective test. YCSB was deployed on an *m1.xlarge* instance. Both YCSB and the writer machine of the Consistency Measurement Component as well as the MongoDB master were deployed in the *eu-west-1a* availability zone. We used a simple load balancer strategy for all tests, where requests were always routed to the closest replica. Cassandra clients were configured to use consistency level ONE for all requests.

During each test, we left the storage system at idle for at least 30 minutes before we started the Consistency Measurement Component. After another 30 minutes we then started YCSB running workload 1. When YCSB was done, we again waited for the storage system to stabilize before running workload 2. Finally, after completing workload 2, we asserted that the system stabilized again at the levels before each workload. This resulted in about 1000 to 1300 writes of the CMC per benchmark for which we measured our consistency metrics.

There were no cross effects between the three different tests as we started each storage system cluster from scratch. Both workloads comprised one million operations on 1000 records. Workload 1 had 80% reads and 20% writes, while workload 2 was configured the other way around.

4.2 Results

Effects of Workload Surprisingly, the workloads barely affected the inconsistency window (t-visibility) of both systems. We used Amazon CloudWatch to also measure the CPU utilization and network IO of the replicas and the YCSB instance. In all cases network IO of the "master" replica[12] seemed to be the bottleneck. During one benchmark, while we were still testing the setup of our scripts, we managed to overload the CPU of Cassandra's "master" replica. During that period we observed very high staleness values. Obviously, when the CPU is saturated, the consistency behavior becomes completely unpredictable. Table 1 shows the CPU utilization that we encountered during our experiments.

During one of the tests (Cassandra in the multi-region setup), we were able to see an effect of the workloads on the inconsistency window. Figure 3 shows how staleness values changed over time during that experiment (the graph shows a moving average to remove extreme values). The boxes indicate the periods during which the two workloads were running.

[11] On AWS, availability zones describe completely independent data centers located next to each other within the same geographical region. AWS regions each have at least two availability zones and are geographically distributed.

[12] The load balancing strategy that we chose effectively asserted that all updates originated on the same replica.

		Workload			
System	**Replica Type**	**idle**	**CMC only**	**read-heavy**	**update-heavy**
Cassandra	Update Coordinator	<5%	ca. 20%	70-80%	70-80%
	Other Replica	<5%	15-20%	ca. 25%	25-40%
MongoDB	Master	<5%	20%	ca. 25%	35-40%
	Slave	<5%	5-10%	ca. 25%	35-40%

Table 1: CPU Utilization During Consistency Benchmarks

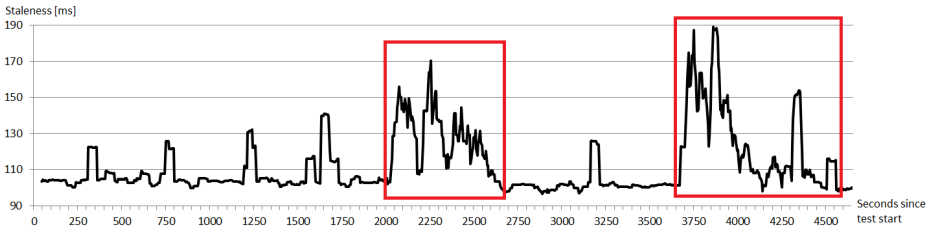


Fig. 3: Change of Staleness over Time (Cassandra, Multi-region Setup)

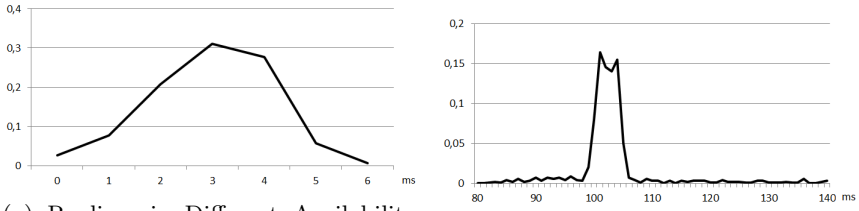
4.3 Effects of Geo-Distribution

For Cassandra, about 98% of all requests created an inconsistency window between zero and one milliseconds when deployed within a single availability zone. As there was only a single maximum value of 38ms, we do not show a chart for this. For the setups where replicas were distributed over three availability zones or regions respectively, Figure 4 shows the observed density functions for the inconsistency windows. We have excluded extreme values from our results to increase clarity of the chart. As expected, it is fairly obvious that increasing the level of geo-distribution increases staleness. We did not encounter any violations of MRC, MWC or RYWC which is caused by both the load balancing strategy that we chose (routing requests to the closest replica) as well as the fact that our benchmarks did not encounter any obvious failures.

For MongoDB, the results were slightly different. As expected, the setup with replicas distributed over different regions showed the longest inconsistency window. We would have expected to see again a value of close to zero for the single availability zone setup and a slightly larger value for the setup in multiple availability zones. Interestingly though, this was exactly the other way around. See Figure 5 for the density functions of observed inconsistency windows on MongoDB.

When looking at the detailed results for the individual replicas[13], it becomes obvious that it was always the same replica that was lagging behind. When we excluded this replica, results are again as expected: More than 96% of all

[13] We do not report those detailed results here due to space limitations, but the CMC logs the result of every single datastore interaction as well as the corresponding timestamp and latency.



(a) Replicas in Different Availability Zones

(b) Replicas in Different Regions

Fig. 4: Distribution of Inconsistency Windows in Cassandra

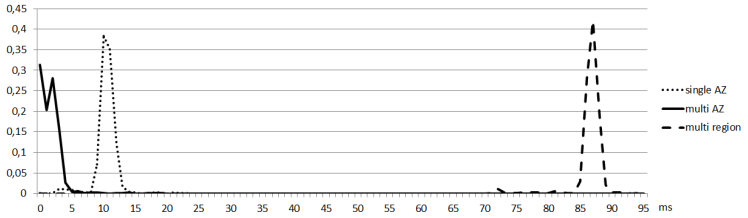


Fig. 5: Distribution of Inconsistency Windows in MongoDB

requests show an inconsistency window of 5ms or less in the single availability zone setup. We believe that this could be caused by one of two effects which are both related to problems with the respective virtual machine. Either the third replica had a problem (possibly due to a resource-greedy tenant on the same physical machine) and was really lagging behind or the CMC reader for this replica had a clock synchronization issue which caused its clock to lag by around 10ms behind. Normally, this should not be an issue as our CMC component was started about 24 hours in advance to allow for a slow clock synchronization process[14]. In this case, one possible reason for causing this effect is a problem with the virtual machine of the CMC reader. However, further investigation is required to verify if other reasons could be behind this effect.

During our multi-region tests with both Cassandra and MongoDB, we could observe that the Singapore region usually added another 15 to 20ms to the inconsistency window already caused by the *us-west* replica. Obviously, the connection to the Singapore replica was the limiting factor in our setup.

4.4 Additional Observations

For Cassandra, we also repeated a multi-region setup with a fourth replica in the region *sao-paulo* and varied the write consistency level of Cassandra which describes the number of replicas that need to acknowledge a write request so that it terminates successfully. In all of our tests, we could not see any variance in the staleness levels due to the write consistency level chosen. Obviously, the

[14] ntp.org recommends about 4 hours, so we really played it safe here.

write consistency level is rather a durability level than a consistency level as the system does not block dirty reads. This implies that in a geo-distributed setting the updates might be visible on some replicas before the request commits at the coordinator of the write which, in essence, corresponds to something like "negative staleness". Apart from increased request latency there was no effect on the system.

5 Related Work

Several studies have been presented as an attempt to quantify the consistency guarantees of cloud storage services. Wada et al. [26] presented an approach for measuring time-based staleness by writing timestamps to a key from one client, reading the same key and computing the difference between the reader's local time and the timestamp read. However, this approach is very primitive and imprecise and is, hence, unsuitable in a production environment. In particular, systems often use a certain degree of sessions stickiness so that most inconsistencies will never become visible to the single client. Arguably, a more complex interaction pattern between benchmarking client and datastore could also be interesting. These limitations hurt the accuracy and meaningfulness of the reported measurements. Bermbach and Tai [5] have addressed parts of these limitations by extending the original experiments of [26] using a number of readers which are geographically distributed. They measure the inconsistency window by calculating the difference between the latest read timestamp of version n and the write timestamp of version $n + 1$. Their experiments with Amazon S3 showed that the system frequently violates monotonic read consistency and exposes very high degrees of staleness. Using the individual reader's read timestamps their approach also allows to easily describe monotonic reads violations as well as the probability of reading fresh or stale data (including the degree of staleness) as a function of the duration since the last update. The accuracy of their measurements in contrast to the single reader-writer setup, though, is limited by the accuracy of the clock synchronization protocol used.

Anderson et al. [2] and Golab et al. [16] presented an offline algorithm and its online analysis extension that builds a dependency graph based on the clients' operation logs and searches for cycles in that graph. Their approach allows to check for violations of safety, regularity and atomicity which are properties developed by the theoretical distributed systems community. It is unclear what the implications of their results are for both system providers (data-centric view) or application developers (client-centric view). Rahman et al. [21] have presented a first step towards defining a standard consistency measurement benchmark and extended their previous work to also consider, e.g., Δ-atomicity and k-atomicity. k-atomicity describes an atomic execution where a maximum version lag of k units could be observed. Δ-atomicity does the same for time. We believe that these metrics are insufficient for benchmarking consistency guarantees of cloud storage systems for several reasons: First, these metrics are very coarse-grained in that they just return the single maximum inconsistency value which could be

observed. For example, in the results of [5] only the highest measurement spike would be reported. Second, although these metrics are from a client perspective, it is unclear how they might be helpful to an application developer. Third, the measurements are highly dependent on the client workload and are, thus, likely to be not reproducible. We believe that their approach is, hence, more suitable for monitoring a consistency health status for a production application where it may be necessary to react to severe consistency violations whereas for benchmarking purposes more detailed metrics are needed which provide meaningful information to application developers.

Zellag and Kemme [27] have proposed an approach for real-time detection of consistency anomalies for arbitrary cloud applications accessing various types of cloud datastores in transactional or non-transactional contexts. In particular, the approach builds the dependency graph during the execution of a cloud application and detect cycles in the graph at the application layer and independently of the underlying datastore. One of their main assumptions though, that of a causally consistent datastore, makes it impractical to use with today's eventually consistent storage systems. We expect future extensions to resolve this issue.

Bailis et al. [3] presented an approach that provides expected bounds on staleness by predicting the behavior of eventually consistent quorum-replicated data stores using Monte Carlo simulations and an abstract model of the storage system including details such as the distribution of latencies for network links. In general, predicting staleness, if accurate, can be used in a variety of ways, such as performance tuning, monitoring system service level agreements and feedback control. Still, a simulation approach is inherently limited in its accuracy as it is only an approximation based on the influence factors considered within the model. Furthermore, PBS is limited to Dynamo-style quorum systems and, thus, not applicable to systems like MongoDB.

Patil et al. [20] also propose to measure staleness in terms of time. Their benchmarking approach, though, can only serve as a rough approximation for consistency as it is subject to the same limitations as the approach described by Wada et al. [26] and also incurs additional inaccuracies due to the way values are measured.

6 Conclusion

In this paper, we presented the first steps for building a standard comprehensive benchmark for quantifying the consistency guarantees of cloud-hosted storage systems. We identified meaningful and fine-grained continuous metrics, the main challenges and requirements for such a benchmark and proposed an architecture for a corresponding benchmarking system. Afterwards, we showed how a comprehensive benchmarking tool could be built reusing proven, standard components. We then used this benchmarking tool to evaluate the effects of geo-replication and different workloads on two popular NoSQL systems, Cassandra and MongoDB, and also studied how different write quorums in Cassandra affect consistency.

In future work, we plan to also include a Tenant Simulator and a Failure Injector, as outlined in section 3, and use it to study the effects of various kinds of failures as well as cross-tenant effects on consistency guarantees of eventually consistent storage systems. We also plan to run additional benchmarks on other storage systems in all kinds of consistency benchmark setups using the components presented within this work. Furthermore, we intend to continue our efforts towards a standardized comprehensive consistency benchmark comparable to performance benchmarks like TPC-W.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer 45(2) (2012)
2. Anderson, E., Li, X., Shah, M.A., Tucek, J., Wylie, J.J.: What consistency does your key-value store actually provide? In: HotDep (2010)
3. Bailis, P., Venkataraman, S., Franklin, M., Hellerstein, J., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. PVLDB 5(8) (2012)
4. Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Léon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: Proc. of CIDR. pp. 223–234 (2011)
5. Bermbach, D., Tai, S.: Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (2011)
6. Bermbach, D., Kuhlenkamp, J.: Consistency in distributed storage systems: An overview of models, metrics and measurement approaches. In: Proceedings of the International Conference on Networked Systems (NETYS). Springer (2013)
7. Bermbach, D., Kuhlenkamp, J., Derre, B., Klems, M., Tai, S.: A middleware guaranteeing client-centric consistency on top of eventually consistent datastores. In: Proceedings of the 1st International Conference on Cloud Engineering (IC2E). IEEE (2013)
8. Binnig, C., Kossmann, D., Kraska, T., Loesing, S.: How is the weather tomorrow?: towards a benchmark for the cloud. In: Proceedings of the Second International Workshop on Testing Database Systems (2009)
9. Bodík, P., Fox, A., Franklin, M.J., Jordan, M.I., Patterson, D.A.: Characterizing, modeling, and generating workload spikes for stateful services. In: SoCC (2010)
10. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC (2000)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26(2) (2008)
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. ACM (2010)
13. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Googles globally-distributed database. To appear in Proceedings of OSDI p. 1 (2012)
14. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP (2007)

15. Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., Markl, V., Tosun, C.: Benchmarking in the Cloud: What it Should, Can, and Cannot Be. In: TPCTC (2012)
16. Golab, W., Li, X., Shah, M.: Analyzing consistency properties for fun and profit. In: Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing. pp. 197–206. ACM (2011)
17. Gray, J. (ed.): The Benchmark Handbook for Database and Transaction Systems (1st Edition). Morgan Kaufmann (1991)
18. Klems, M., Bermbach, D., Weinert, R.: A runtime quality measurement framework for cloud database service systems. In: Proceedings of the 8th International Conference on the Quality of Information and Communications Technology. Springer (2012)
19. Lakshman, A., Malik, P.: Cassandra: A structured storage system on a p2p network. In: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. pp. 47–47. ACM (2009)
20. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In: Proceedings of the 2nd ACM Symposium on Cloud Computing. p. 9. ACM (2011)
21. Rahman, M.R., Golab, W.M., AuYoung, A., Keeton, K., Wylie, J.J.: Toward a Principled Framework for Benchmarking Consistency. In: HotDep (2012)
22. Sakr, S., Liu, A., Batista, D.M., Alomari, M.: A Survey of Large Scale Data Management Approaches in Cloud Environments. IEEE Communications Surveys and Tutorials 13(3), 311–336 (2011)
23. Silberstein, A., Chen, J., Lomax, D., McMillan, B., Mortazavi, M., Narayan, P.P.S., Ramakrishnan, R., Sears, R.: PNUTS in Flight: Web-Scale Data Serving at Yahoo. IEEE Internet Computing 16(1) (2012)
24. Tanenbaum, Andrew S. ; Steen, M.v.: Distributed systems : principles and paradigms. Pearson, Prentice Hall, Upper Saddle River, NJ, 2. ed. edn. (2007)
25. Vogels, W.: Eventually Consistent. Queue 6 (October 2008), http://doi.acm.org/10.1145/1466443.1466448
26. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In: CIDR (2011)
27. Zellag, K., Kemme, B.: How Consistent is your Cloud Application? In: SoCC (2012)
28. Zhao, L., Sakr, S., Fekete, A., Wada, H., Liu, A.: Application-Managed Database Replication on Virtualized Cloud Environments. In: Data Management in the Cloud (DMC), ICDE Workshops (2012)
29. Zhao, L., Sakr, S., Liu, A.: Application-Managed Replication Controller for Cloud-Hosted Databases. In: IEEE CLOUD (2012)