Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies

David Bermbach
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: david.bermbach@kit.edu

Stefan Tai
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: stefan.tai@kit.edu

***Abstract*—Cloud storage services and NoSQL systems typically guarantee only *Eventual Consistency*. Knowing the degree of inconsistency increases transparency and comparability; it also eases application development. As every change to the system implementation, configuration, and deployment may affect the consistency guarantees of a storage system, long-term experiments are necessary to analyze how consistency behavior evolves over time. Building on our original publication on consistency benchmarking, we describe extensions to our benchmarking approach and report the surprising development of consistency behavior in Amazon S3 over the last two years.**

Based on our findings, we argue that consistency behavior should be monitored continuously and that deployment decisions should be reconsidered periodically. For this purpose, we propose a new method called *Indirect Consistency Monitoring* which allows to track all application-relevant changes in consistency behavior in a much more cost-efficient way compared to continuously running consistency benchmarks.

I. Introduction

Over the last few years, cloud storage services and self-managed NoSQL systems deployed on cloud compute services have found widespread adoption. To cope with spiky workloads and the resulting elastic scalability requirements, low latency and high availability demands, these systems tend to relax consistency guarantees as required by the consistency vs. availability and consistency vs. latency tradeoffs [1], [2]. This resulted in a situation where most of these systems offer only *Eventual Consistency* instead of strict consistency combined with rich transactional guarantees. While this is attractive from a storage system design perspective, it shifts complexity to the application developer. On the other hand, developing applications on top of eventually consistent storage systems can be done for many classes of applications [3], [4]; still knowing the degree of inconsistency is helpful to an application developer as the dimension of uncertainty is reduced: The quality of service (QoS) guarantees become, to a degree, predictable. Also, knowing about consistency behavior allows to increase the efficiency of middleware solutions increasing consistency guarantees outside of the storage system [5].

For QoS dimensions like latency or throughput, benchmarking approaches, e.g., the Yahoo! Cloud Serving Benchmark (YCSB) [6], have been studied for a long time. In the case of consistency guarantees and behavior, though, benchmarks and measurement approaches have only been developed over the last few years [7]–[13]. More recently, Bailis et al. [14] presented an approach to also predict staleness behavior of certain classes of storage systems.

In our original paper on consistency benchmarking in 2011 [8], we chose Amazon S3[1] as an example and studied its consistency behavior. For S3, Amazon guarantees only Eventual Consistency but we were interested in the actual degree of inconsistency visible to applications. In our experiments, we discovered a rather curious behavior of periodically recurring patterns. After publication, we also contacted Amazon about this topic; while they would not comment on the root source of the effects, they quickly made changes to the service resulting in a different consistency behavior. Over the last two years, we continued to benchmark S3 periodically as a long-term experimental study and now present the extensive changes we could see during that period.

Based on our observations, we argue that it is necessary for application developers and operators to continuously monitor QoS levels of cloud storage services and to periodically reconsider the deployment decisions as the customer is entirely at the provider's mercy. For this purpose, we present a new method called *Indirect Consistency Monitoring* which can be used to monitor consistency behavior in a cost-efficient way while being able to track all changes in consistency behavior that actually affect the application running on top of it.

The remainder of this work is structured as follows: We start with a few definitions and recap the main ideas of our consistency benchmarking approach in section II. We also describe a set of extensions that we added over the last two years. Afterwards, in section III we describe the setup and results of our long-term experimental study with S3 before discussing our observations as well as the resulting implications in section IV. Based on the lessons learned from our long-term study, we derive and describe a new method for consistency monitoring in section V. Finally, we show up related work (section VI) before coming to a conclusion in section VII.

II. Consistency Benchmarking

In this section, we will start with the definition of our understanding of consistency. Afterwards, we will recap our staleness measurement approach before discussing our extensions of it.

[1] aws.amazon.com/s3

A. Consistency Definitions, Perspectives and Dimensions

In distributed storage systems, there are two competing definitions of consistency: From the perspective of the database community, consistency refers to the "C" in ACID[2], i.e., the adherence to integrity rules and a data schema. The distributed systems perspective, which we will take for the remainder of this work, is fundamentally different, though: a datastore is in a consistent state if all replicas are identical and all ordering guarantees of the corresponding consistency model are observed [15], [16]. Originally, this definition requires all data items to be identical. In practice, though, guarantees are typically only provided per key [16].

Based on this, *Eventual Consistency* is a rather weak consistency model not requiring any ordering guarantees and only offering "eventual" replica convergence. I.e., an eventually consistent storage systems guarantees that the system will – in the absence of failures and further updates – eventually converge towards a consistent state where all replicas are identical [16]–[18].

As can be seen in the definition of consistency, there are two dimensions of consistency: *Staleness* and *Ordering*. While Staleness describes how "far" replicas are apart (either in terms of time or missing versions), Ordering describes to which degree incoming requests may be reordered on different replicas. For instance, *Sequential Consistency* requires that all replicas must execute all requests in the exact same order whereas *Causal Consistency* requires this only for (potentially) causally related operations.

There are also two perspectives on consistency: a data-centric and a client-centric view. While the data-centric view corresponds to the perspective of a provider, describing requirements on synchronization processes (e.g., "all replicas must agree on a total order of updates before applying the updates"), the client-centric view is focused on the consistency effects visible to the client of a storage system (this may, for instance, be an application server). An example of such a guarantee is *Monotonic Read Consistency* (MRC) which requires that a client will, after reading a version n, never again read a version older than n [19]. We propose to use the client-centric perspective for measurements which is common practice for other QoS dimensions, e.g., latency and throughput.

B. How soon is eventual?

Based on [8], [9], staleness and MRC can be measured by periodically writing to a storage system while continuously reading the same key. The difference in time between the last read of version n and the write timestamp of version $n + 1$ determines the client-centric t-Visibility [14] for systems with dirty reads (which is the normal case for cloud storage services and NoSQL systems).

As storage systems often use sticky sessions or route requests to the respectively closest replica, this does not guarantee that a comprehensive view spanning multiple clients can be achieved in measurements. For this reason, in [8] we extended [9] to use dedicated, geographically distributed machines for reading at the same time, figure 1 shows an abstract setup during benchmarks. The number of readers also depends on the number of replicas as – under the assumption that all readers issue a request at the same time and that the load balancer randomly routes requests to replicas – the probability of reaching all replicas is a function of the number of readers. An optimal number of readers can then easily be calculated.

[2]Atomicity, Consistency, Isolation, Durability.

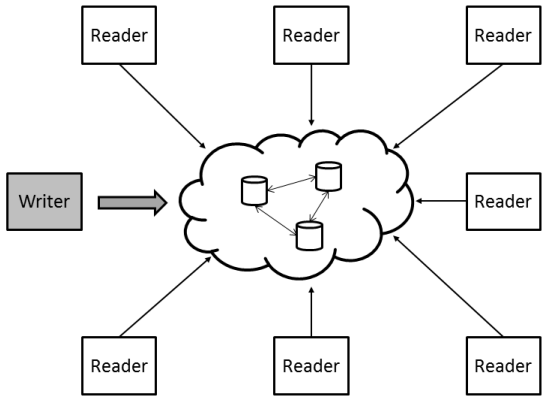

Fig. 1: Setup During Staleness and MRC Measurements

During a benchmark the writer periodically writes a tuple *(writeTimestamp;version)* to the storage system. All readers record detailed logs for each read containing tuples *(readTimestamp;writeTimestamp;version)*. An offline analysis (or an online analysis lagging slightly behind) can then determine MRC violations in each individual reader's logs as well as the maximum inconsistency window, the probability of stale vs. fresh reads, and the version lag (k-Staleness [14]) based on all reader logs.

Recently, we also added functionality to determine staleness for delete operations, i.e., how long a version is still readable after issuing a delete request. In this case, a set of machines creates a workload on the respective key – the kind (read/write) and intensity of the workload could potentially influence the staleness for delete operations – until it is deleted by the writer. Then the readers continuously issue read requests for the same key. Based on the delete timestamp and the timestamps when the deleted version could or could not be read, it is then possible to determine a function describing the probability of reading a value as a function of the time since deleting the datum. Of course, this process must be repeated many times to produce statistically significant results.

For geo-distributed storage systems, comparably distributed readers are used, e.g., [7]. Potentially, more than one writer instance (also geo-distributed) can be used whose writes are then interleaved to study how the origin of the update affects consistency behavior.

C. Beyond Staleness Measurements

As results can be obtained by the same experiment setup as staleness measurements, we also described MRC measurements in the last section even though it is an ordering guarantee. Beyond MRC, there are also *Read Your Writes Consistency* (RYWC), *Monotonic Writes Consistency* (MWC), and *Write Follows Read Consistency* (WFRC) as client-centric ordering guarantees. While RYWC requires that after writing

a version n a client will always be able to see the result of his write or a newer version, MWC describes that two subsequent writes by the same client will be serialized in their chronologic order [19]. Note, that time is not part of this concept. While all three guarantees are important to application developers, MWC is especially critical since applications on top of systems without MWC "are notoriously hard to program" [18].

MWC can be measured by having a machine issue two directly subsequent write requests to the same key. While it does not matter which value is returned directly after the second write, the final serialization order must be correct. In practice, it should suffice to wait for at least the duration of the maximum inconsistency window including peak values – in our experiments we checked several times over the course of 24 hours. We propose to repeat these measurements for a large number of keys to achieve statistical significance.

RYWC can be measured by having a machine write a value and continuously read the same key for a long time. After a while, the machine can issue another update and so on. Essentially, this corresponds to the setup of the staleness measurements of Wada et al. [9]. This approach provides the necessary data to determine the probability of RYWC violations as a function of time since the last update.

As this workload is rather simplistic, it might make sense to use a more complex workload, e.g., [5]. This approach, though, has the disadvantage that not the RYWC violations caused by the storage system will be measured but rather those caused by concurrent updates. For instance, client A may read a value at time t_0 and write an updated value at time t_1. If client B also updates the value between t_0 and t_1, then the update of B will be lost and B will not be able to read his write back, i.e., RYWC is violated. Hence, such a workload would create RYWC violations even in the case of non-replicated storage systems as long as latencies are > 0. As these violations are not caused by the storage system but instead are a function of the access pattern and number of clients, we propose to use the more simplistic workload with just one machine to only capture RYWC violations actually caused by the storage system.

For the fourth client-centric ordering guarantee, WFRC, we have not been able, yet, to find a way of measurement. See also [16] for a more detailed discussion of the problems in measuring this behavior.

III. Experiments

In this section, we present the results of our long-term experimental study using Amazon S3 as an example. While the actual measurement results are obviously S3-specific, the methods used are independent of concrete storage systems. We also expect comparably unpredictable changes in consistency behavior for other cloud storage services and NoSQL systems.

We will start by describing the results our staleness measurements over time before continuing to other measurement results.

A. Staleness and MRC Results

For each of our experiments, we deployed 12 readers on EC2 small instances[3] in the AWS region eu-west, 4 each in every availability zone; we also deployed a writer instance in zone A. We did so as S3 keeps replicas within a region but distributed over several availability zones. In our experiments, we varied both the test duration (24 hours or 7 days) and the interval between updates (10 or 20 seconds), see table I.

In experiment 1 – which we actually repeated several times with the same behavior – we observed the two periodicities already reported in our original paper [8]: Alternating LOW and SAW phases which we later determined to be correlated to working hours since the experiment started at Monday morning, 10.30h local time. Figure 2 shows how the maximum observable t-Visibility values change over time, the chart has been clipped at 20 seconds of staleness excluding peak values of up to almost 35 seconds. During the SAW phases ("black areas" in figure 2) staleness values changed in a sawtooth pattern with a wavelength of slightly below two minutes. Figure 3 shows a randomly selected excerpt from a SAW phase. The rest of the time, i.e., during the LOW phases, values were as expected distributed randomly. We believe, also based on experiments where multiple files were targeted with benchmarks at the same time, that the S3 implementation at that time initiated update propagation either periodically or upon a second write to the same bucket.

On October 20, 2011, we contacted Amazon regarding this behavior. While they would not comment on the root source of this effect, they quickly made changes to their implementation which we were able to observe in experiments 2, 3 and 4: While the daily patterns were still clearly visible (see figure 4, the chart has been clipped at 25 seconds) and the LOW phase had not changed, the SAW phase showed an "obscurified" but still recognizable sawtooth pattern, see figure 5 for a randomly selected excerpt from the SAW phase; experiments 2, 3, and 4 showed almost identical results. Interestingly, the median of the staleness values increased so that the update actually resulted in poorer consistency behavior.

Almost one year later, we again benchmarked S3 – a 24 hour experiment and another 7 day experiment to verify our findings. Amazon must have made additional changes to their system as the daily/weekly patterns of SAW and LOW phases did not exist any longer. Furthermore, even maximum peak values did not exceed 10 seconds any more; in fact, most peaks are below one second. Figure 6 shows the results of experiment 6 (experiment 5 showed almost identical behavior); the chart has been clipped at 500ms – still, we could count less than 50 peaks of more than one second out of 60,000 measurements. This was actually the behavior we would have expected for our original experiments in 2011.

Another year later, in September 2013, we repeated experiments 5 and 6. During these experiments we observed that the number of peaks had increased dramatically., Furthermore, even though these peaks were typically between 6 and 10 seconds, there were also quite a few peaks still exceeding 10 seconds of staleness. Average values increased by about 100ms, median values doubled and standard deviation values

[3] aws.amazon.com/ec2

increased between 300% and 600% compared to experiments 5 and 6. While results are still far better than in 2011, they are now, in 2013, effectively worse than in 2012. Especially, the increase in variance has caused the consistency behavior to become much more unpredictable. Figure 7 shows the results of staleness measurements in experiment 7. When comparing figures 6 and 7, note that figure 7 uses a logarithmic scale..

Figure 8 and table II show how the aggregated results have evolved over time.

While analyzing the results of experiment 8, we also discovered a curious "bump" in the chart showing the probability of reading non-stale data as a function of time since the last update (see figure 9). When we went back to also recheck the results of the other 7 experiments, we realized that they all had the same "bump" in the very same time interval. Further analysis revealed that there is also a temporary increase in average read latency values right before the "bump" (see figure 10 which shows the read throughput as well as the average read latency over time). As this behavior keeps reoccurring in all 8 experiments of which experiment 8 alone is based on more than half a billion reads, we believe that this is not random behavior.

Obviously, the increased read latency affects only fresh reads while stale reads continue unaffectedly. The only explanation that we can come up with, is that about 30ms after an update started (we have single digit update latencies), the only existing fresh replica blocks briefly to update another replica, while a third (still stale) replica continues to serve requests.

Recently, we also extended our benchmarking approach to measure the inconsistency window of delete operations, i.e., how long a value is still readable after having been deleted. In the results of several experiments, we noted that there is no influence of the workload before the delete on staleness of deletes which indicates that Amazon uses a constant number of replicas independent of the actual workload on a specific key. It is, therefore, also unlikely that Amazon uses caching layers in S3. The results of our delete experiments were comparable to the results of experiments 7 and 8 so that it seems likely that delete and update operations are implemented in a similar way.

As already mentioned, our approach for staleness measurements also logs the results of MRC violations and read error rates. Table III gives an overview of the results. Note, that the MRC results of experiments 2 and 3 have to be normalized as most updates complete below 10 seconds so that during the second half of the update interval MRC guarantees can only be violated if there is a peak value with a staleness value beyond 10 seconds. A rough approximation of the normalized results is to multiply the measured values with a factor of two, thus, simply excluding all staleness values beyond 10 seconds.

Based on these results, we can see that the original Amazon update slightly improved MRC behavior before really driving the rate of MRC violations down in 2012 where we also had the best staleness results. In our 2013 experiments, we then again saw an increase in MRC violations – while the values continue to be low, they still increased by a factor of approximately 30 between 2012 and 2013.

TABLE I: Experiment Setups

Experiment	Start Date	Duration	Write Interval	Commment
1	Aug 29, 2011	7d	10s	-
2	Nov 10, 2011	24h	20s	one reader crashed
3	Nov 16, 2011	24h	20s	-
4	Nov 17, 2011	24h	10s	-
5	Oct 2, 2012	24h	10s	-
6	Oct 3, 2012	7d	10s	-
7	Sept 4, 2013	24h	10s	-
8	Sept 6, 2013	7d	10s	-

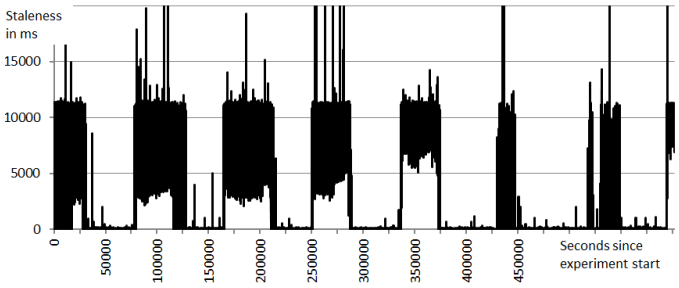

Fig. 2: SAW and LOW Phases in Experiment 1

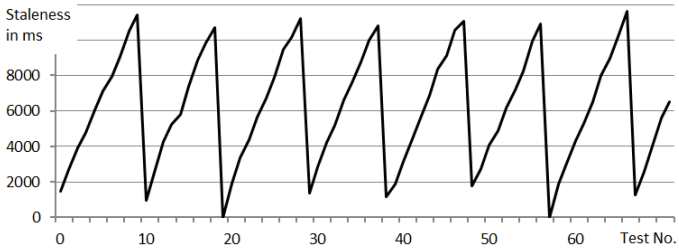

Fig. 3: Excerpt from a SAW Phase in Experiment 1

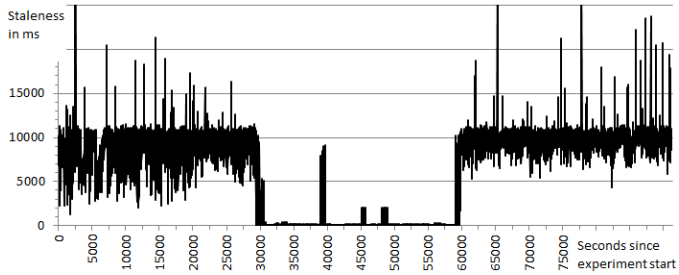

Fig. 4: SAW and LOW Phases in Experiment 4

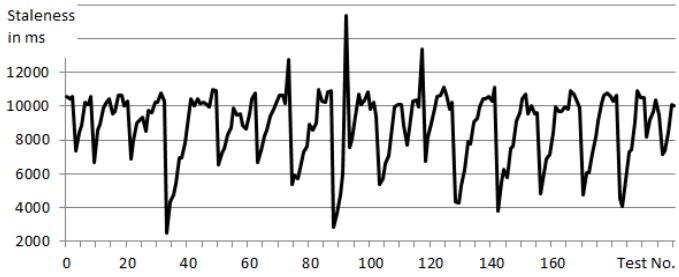

Fig. 5: Excerpt from a SAW Phase in Experiment 4

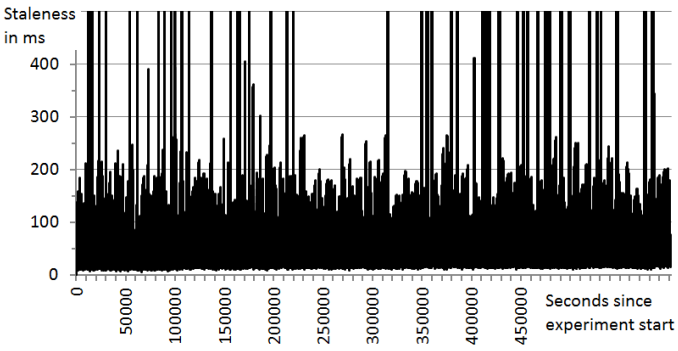

Fig. 6: Staleness Results of Experiment 6

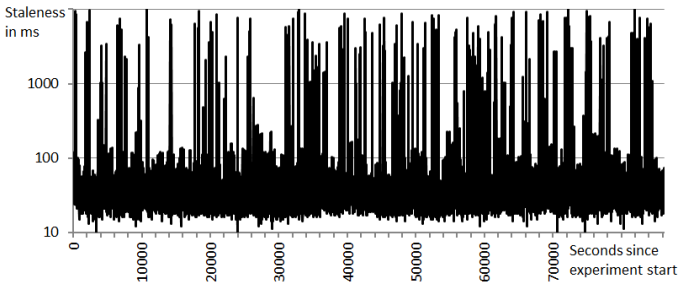

Fig. 7: Staleness Results of Experiment 7

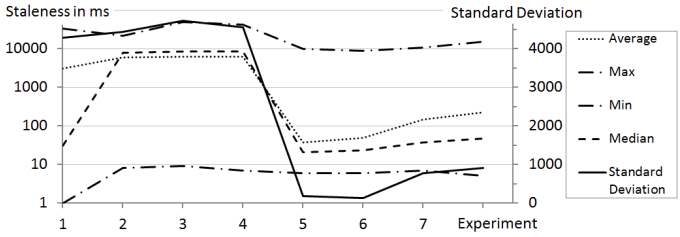

Fig. 8: Overview of Results in Experiments 1-8

TABLE II: Results of Experiments 1-8

Experiment	Min.	Avg.	Median	Max.	Std. Dev.
1	1	3,029.48	31	32,716	4,276.60
2	8	6,072.17	7,969	21,439	4,431.88
3	9	6,220.37	8,603	48,281	4,731.92
4	7	6,277.14	8,648.5	42,922	4,553.42
5	6	36.65	21	9,854	183.62
6	6	48.81	23	8,663	132.10
7	7	147.77	37	10,594	767.56
8	5	223.14	47	15,165	908.35

TABLE III: MRC and Error Results of Experiments 1-8

Experiment	% Prob. of MRC Violations	Prob. of Read Errors
1	12.0461%	$2.8 * 10^{-9}$
2	4.8603%	$7.2 * 10^{-7}$
3	5.0634%	$3.6 * 10^{-5}$
4	10.6982%	$6.8 * 10^{-5}$
5	0.0013%	$3.1 * 10^{-7}$
6	0.0005%	10^{-7}
7	0.0359%	$5 * 10^{-8}$
8	0.0364%	$7.3 * 10^{-8}$

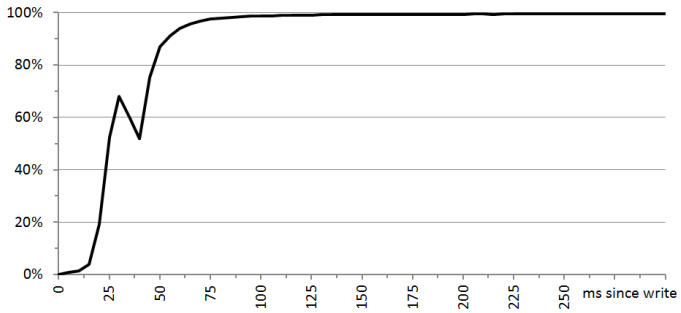

Fig. 9: Probability of Reading Fresh Data as a Function of the Time since the Last Update (Experiment 8)

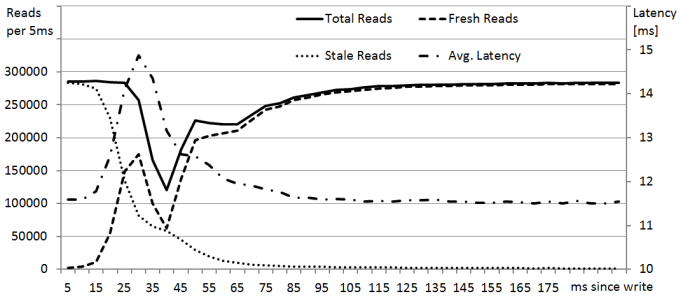

Fig. 10: Influence of the Time since the Last Update on Latency and Read Rates (Experiment 8)

B. *Further Results*

We also benchmarked MWC and RYWC violations. For this purpose, we deployed a single EC2 small instance in the Amazon region eu-west. For the MWC measurements, this instance issued an update and directly afterwards issued another update to the same key. We did this for 1000 keys. Afterwards, we periodically checked over the course of 24 hours whether the value that could be read was identical to the value of the second update. Neither of these checks returned the value of the first update so that the result of our experiment was, hence, that MWC seems to be guaranteed by S3. We ran this benchmark on March 19, 2012 and September 4, 2013.

On the same dates we also ran our RYWC benchmark where we first issued a write and then read this value back 100 times. This was repeated for 1000 writes and never read any value older than the one our client had just written. We, therefore, conclude that S3 also seems to guarantee RYWC. Of course, both conclusions are only based on the random sample we observed which does not preclude that violations may occur from time to time.

IV. Discussion

In our experiments, we have seen high variance in the results over time caused by changes which will not be immediately noticed by a cloud storage customer. This can potentially cause high unexpected cost to the cloud storage customer as can be seen in the following example business case:

Imagine a scenario where a web shop has implemented its stock management application on top of S3. Both check-out operations and operations logging delivery of new products

will then concurrently update the same key in S3. In this case, the rate of overselling could have suddenly increased both in 2011 as well as in 2013 caused by the changes visible in experiments 2 and 7 respectively. In both cases, average staleness unexpectedly increased which would lead to higher probabilities of reading stale data and, thus, to an increased rate of overselling. Still, this could have been easily handled by simply using a larger quantity of items stored per product *if* the change in behavior had been known beforehand. If now the price for having one more item in stock is lower than the compensation cost for overselling, then the web shop would have incurred a large jump in almost entirely avoidable compensation cost.

On the other hand, when staleness results heavily improved in 2012 (as seen in experiment 5), the web shop would have been paying too much for inventory cost as the safety margin for the quantity of items in stock could have been much lower. Knowing about consistency behavior, hence, often directly affects the budget of a cloud storage-based company. Furthermore, while weak consistency guarantees may be a problem, changes in consistency behavior are even more problematic.

The problem here is that concrete consistency results are typically not included in service level agreements (SLAs) – to our knowledge no cloud storage provider is doing so. Therefore, consistency behavior of a cloud storage system is entirely unknown to the customer and may also, as we have seen, frequently change without advance notice. For all scenarios where consistency violations directly cause costs to the customer, we, hence, propose to continuously monitor these costs and their root sources[4]. If there is a sudden change, we then propose that a cloud storage customer (re-)benchmarks the system to identify where the effect is coming from. Based on these results, appropriate action can be taken, e.g., changing implementation details or business details like the minimum stock amount in our example, contacting the provider about the problem, or even switching the cloud storage provider.

Contacting the provider is almost always helpful since he might – especially for more "unusual" quality dimensions like consistency – not even know about the undesired side effects of a change in implementation. If the customer is "big" enough, e.g., Netflix[5] in the case of AWS, he might even have the necessary influence to persuade the provider to either include the quality in SLAs or to make additional changes to the implementation.

Furthermore, we already mentioned changing the provider as a last resort which may come with problems of its own: The lack of standardized cloud storage interfaces and varying QoS levels [20] leads to a high degree of vendor lock-in where the customer is tied to a specific provider's offering. Suitable mechanisms like Cloud Federation [21]–[27] should be considered to reduce the degree of vendor lock-in and, thus, to maintain the flexibility of provider selection.

Finally, using a self-managed NoSQL system running on top of cloud compute services might in the end prove to be the more economical choice over a hosted storage service.

[4]In our example above, the increase of the overselling rate could also have been caused, e.g., by a change in shop customer behavior resulting in a higher demand for said product.

[5]netflix.com

V. Indirect Consistency Monitoring

As just discussed, our results have several implications for a cloud storage customer: He could promote and demand the development and adoption of cloud standards, as well as use cloud federation to maintain the flexibility of provider choice. He also could ask for consistency guarantees to be part of the SLAs – still, the customer would need to trust the provider to fulfill the guarantees that were promised. Alternatively, he could use a NoSQL system running on top of (potentially private) cloud compute instances, but even here the application is not safe from sudden changes in consistency behavior which may, for instance, be caused by small changes to the network routing settings within the datacenter.

All these implications, may they be about contracts or the flexibility to make changes to the deployment, have the same requirement: continuously updated information on consistency behavior, i.e., running a consistency *monitoring* component. While it may be tempting to deploy our consistency benchmarking approach and keep it running permanently while tracking the measurement results for sudden changes or long term trends, this will usually not be feasible due to the cost it causes. Roughly calculated, continuously running our S3 benchmarks as described in section III causes annual costs of slightly below 20,000 USD[6]. While this amount may still be worth the expense depending on the potential compensation costs caused by inconsistencies, a continuous consistency benchmarking is no longer reasonable when multiple storage services or NoSQL systems running in various setups in several regions are used. For private cloud deployments of NoSQL systems, similar costs will be caused even though they will be less directly visible.

For an economically feasible consistency monitoring solution, we propose to use what we call *Indirect Consistency Monitoring* – instead of measuring consistency behavior, we propose to monitor KPIs (key performance indicators) that are both directly affected by inconsistencies and less expensive to track. Whenever one of those KPIs changes, we propose to rerun our consistency benchmark, thus, significantly reducing the cost of consistency monitoring as benchmarks are only triggered when necessary. *Indirect Consistency Monitoring* comprises four steps before it is actually up and running. We will describe these in the following.

A. Step 1: Identify Datastore Interaction Patterns

In this step, the application source code is analyzed for what we call *datastore interaction patterns* which might be implemented as transactions or business processes in more traditional enterprise systems. Such a datastore interaction pattern comprises the following information: Where in the source code is which data accessed in which way, i.e., which sequence of reads and writes to which key is triggered by which business operation.

For instance, a check-out operation in a web shop might insert order data under a new key and update (i.e., read and overwrite) the amount on stock for each ordered product under already existing keys.

[6]7971.60 USD for instances; 1.14 USD for S3 storage; 15.77 USD for writes to S3; 11,720.69 USD for reads from S3

B. Step 2: Identify Potential Conflicts Between Patterns

In this step, the set of datastore interaction patterns is analyzed to identify all patterns that access the same keys so that there might be cross-effects between the corresponding business operations. Furthermore, all patterns need to be found which are in conflict with themselves either by accessing the same key more than once or with parallel instances of the same pattern. Conflicts occur whenever there is concurrent access to the same key with at least one interaction pattern comprising an update of said key.

For instance, the check-out pattern from above is in conflict with itself when two web shop customers order the same product so that the same (product) key is updated. It might also influence a potential recommendation system pattern which analyzes existing order data to recommend products based on the customer's past preferences.

C. Step 3: Identify Affected KPIs

In this step, only datastore interaction patterns with conflicts are of interest since patterns without conflict will usually not be affected by eventually consistent guarantees. For each of those conflicts, the effects of staleness and ordering changes are analyzed: What happens if staleness increases or decreases? What happens if MRC, RYWC and MWC behavior changes? Which business KPIs will be affected?

For instance, changes in staleness behavior for our check-out operation will affect the accuracy of the amount on stock. So, the frequency and intensity of having more or less products physically on stock than the amount persisted within the storage system will be proportional to staleness behavior of the storage system. Obviously, this also affects the frequency of overselling. Hence, all these statistics could be used as a KPI for staleness.

D. Step 4: Identify and Track Suitable KPIs

The result of the last step is a large selection of possible KPIs which could be monitored instead of staleness and ordering behavior. Some of those may be hard to track (e.g., the discrepancy between the amount physically on stock and the amount persisted within the storage system) whereas others may already be monitored for business intelligence purposes (e.g., the amount of compensation cost due to overselling). Therefore, it makes sense to identify KPIs that are already monitored or for which monitoring can be added easily at little or no additional cost. We propose to track at least one KPI for each of the ordering guarantees as well as staleness: Tracking more than one KPI for each guarantee may cause additional monitoring cost but decreases the chance of starting a consistency benchmark due to a KPI being affected by other external effects (e.g., a delayed product delivery for the KPI that describes the rate of overselling).

In our web shop example, the frequency of overselling is a good KPI for staleness. RYWC describes whether a customer is able to see his own writes (e.g., an order). Hence, a good KPI for RYWC is the frequency of a customer issuing the same order twice within a short timespan and canceling one of them afterwards. MRC describes whether a client only sees increasing versions. Therefore, a simple way to track MRC would be to timestamp a frequently changing data item, e.g., the amount on stock, and have the end user's browser cache those timestamps. Whenever a new subpage of the webshop is requested either directly or via AJAX, the information on the sequence of timestamps seen can be piggybacked on top of the HTTP request. The application servers could then directly track the frequency of these MRC violations as a KPI for MRC. An alternative KPI when using a middleware to increase consistency guarantees outside of the storage system [5], would be to track the rate of resolved MRC and RYWC violations within this middleware component which is easily possible. MWC describes whether two sequential updates are serialized in the correct order. If a customer corrects his order during the check-out process, he is vulnerable to MWC violations. A good KPI is then tracking the number of failed credit card charges after the customer changed his credit card data. Of course, depending on the design of the application and the concrete use case, other KPIs will be more suitable.

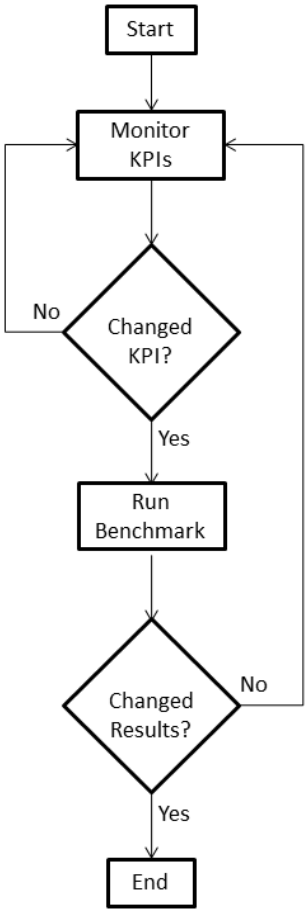

Fig. 11: Flow Chart for Indirect Consistency Monitoring

Figure 11 shows the flow chart of Indirect Consistency Monitoring starting after having identified all KPIs and ending at a point where changes to the consistency behavior have been discovered so that suitable action can be taken.

E. Implementation and Tool Support

Indirect Consistency Monitoring is currently not directly supported with a tool. With a combination of tools, though,

much can be automated: Our consistency benchmarking tool, which is available as open source[7], can be automatically deployed and run by systems like Opscode Chef[8] or [28]. Many KPIs can be tracked by standard monitoring tools like Ganglia[9] or even complex event processing solutions. The remainder can be stitched together using cron jobs and shell scripts.

F. Discussion

Indirect Consistency Monitoring is a way to continuously monitor consistency behavior visible to applications at very little additional cost. It comes with a few caveats, though, which we believe to be for most use cases negligible in comparison to the benefits that are realized:

There may be *false negatives*, i.e., the consistency behavior changes but neither of the KPIs shows any changes. In this case, the application workload obviously hides the inconsistencies. For instance, if maximum staleness changes from one to two seconds but keys are only accessed every ten seconds, then the change in staleness behavior will not be seen by the application. We believe that this is sufficiently precise for most cases as inconsistencies not visible hardly matter to the application. Of course, there is the risk that the consistency behavior will continue to gradually change until inconsistencies become visible so that some kind of pre-warning would be appreciated. This is only possible if there is a KPI that captures these small changes.

There may be *false positives*, i.e., the KPI changes but the underlying consistency behavior did not change. This can be addressed by tracking more than one KPI for each consistency metric, but there is a trade-off between the complexity and cost of KPI tracking and the frequency of running consistency benchmarks.

There is *much effort* for the analysis of source code (our four steps above) before the KPIs can be tracked and some of this effort needs to be repeated at least for major updates to the application. We believe that the analysis becomes feasible if someone familiar with the application's source code is doing it and that KPIs can be chosen in a way so that updates will not require re-analysis of the entire source code, i.e., using KPIs that are based on core functionality which is unlikely to change. In some cases, though, the cost for continuously running our consistency benchmark may be lower, though, than the cost of *Indirect Consistency Monitoring*. This is especially true for MWC and RYWC which can easily be periodically benchmarked with just one machine at very little cost.

All in all, *Indirect Consistency Monitoring* can be a way to track consistency behavior in real world applications in a feasible way. More work is necessary to prove that this idea actually works in practice and is not a mere thought experiment.

Upon a change in consistency behavior, the application provider has several options. In each case, the "best" option depends on the concrete circumstances. For instance, a provider can be contacted or switched, a different configuration option of the storage service may show different consistency behavior, business details (e.g., the quantity stored per product) can be adapted, the implementation of the application can be changed, a self-hosted NoSQL system can be downgraded to a previous version, etc. More work is needed to identify all possible directions of action as well as to methodically select the respectively "best" option.

The work by Rahman et al. [13] could be an alternative – possibly more precise – way to track changes in consistency behavior with the goal of triggering consistency benchmarks only when necessary. However, to our understanding this comes with the price of an additional significant computational overhead. As Rahman et al. themselves point out, their approach needs to constantly solve optimization problems and "[...(they)] are not aware of an efficient (i.e., poly-time) solution to this problem[...]" [13]. Furthermore, to our understanding their calculations of Δ- and k-atomicity require full knowledge on *all* data written and seen by *all* clients, i.e., additional communication with *all* application servers is necessary. Depending on the number of application servers, this can easily become too costly or live analysis may no longer be possible due to the sheer data volume. When trying to use their approach for consistency monitoring, we, therefore, believe that at least steps 1 and 2 of our *Indirect Consistency Monitoring* method need to be executed first to identify suitable application subsets which actually can be monitored using their approach.

Our *Indirect Consistency Monitoring* method is based on the notion that every application provider monitors consistency behavior himself. An alternative could be to have a company offer continuous consistency monitoring with notifications in case of changes as a service to application providers. This may be feasible but comes down to an issue of trust: The application provider needs to trust the monitoring company not to have malicious intent (e.g., getting paid by the cloud storage provider in exchange for a higher reported quality). Furthermore, this alternative becomes infeasible if the cloud storage provider offers different service level classes to different groups of customers.

VI. Related Work

There are several publications on the assessment of consistency behavior.

Wada et al. [9] presented an approach for measuring time-based staleness. During benchmarks, they have one machine issue periodic writes while continuously reading with the same machine. As they use only one machine for the benchmark, their approach is not limited to NTP[10] accuracy but in exchange incurs the disadvantage that it can only report inconsistencies that cannot be hidden by session stickiness. For this reason, they could not see any inconsistencies in their measurements of S3.

In [8], we extended this approach by using multiple machines for reading and writing from and to the storage system. Here, results are limited to NTP accuracy but are no longer vulnerable to load balancer strategies like session stickiness or routing the request to the closest replica. The results from [8]

[7] https://code.google.com/p/consistency-monitoring/

[8] opscode.com/chef

[9] ganglia.sourceforge.net

[10] ntp.org

correspond to the results of experiment 1 in this paper which we reevaluated.

In [7], we extended our approach to a tool suite which is able to also capture changes to effective consistency behavior caused by parallel workloads, geo-replication, multi-tenancy and failures. We used the extended approach to study the effects of geo-replication strategies and different parallel workloads in Apache Cassandra [29] and MongoDB[11].

Anderson et al. [10], Golab et al. [11] and Rahman et al. [13] of HP Labs present an alternative approach using metrics based on Lamport's definitions of safeness, regularity and atomicity [30]. These definitions stem from the synchronization of processes in multi-core systems. As access times to the main memory as well as L1 cache (the same holds for L2, L3, etc.) are very low, staleness values are for multi-core systems negligible which is entirely different in distributed storage systems. Since each of these metrics also muddles staleness and ordering guarantees, they very well fit the analysis of ordering guarantees for multi-core systems where one out of two influence factors on the metric output is close to zero. In the case of distributed storage systems, though, it is entirely unclear which influence factor (staleness or ordering) causes a concrete value. We, therefore, believe that they cannot provide meaningful results – neither to storage providers nor to application developers using said storage system. The authors yet have to show how one can use their results apart from ranking different systems. Additionally, as these metrics are very coarse-grained capturing only maximum deviations, they are best used for systems offering (almost) strict consistency with very low variance which, as we have seen in our experiments, is rarely if ever the case in eventually consistent storage systems.

Zellag and Kemme [31] also present an alternative approach counting consistency anomalies for arbitrary cloud-based applications in transactional and non-transactional datastores. At runtime, their approach builds a dependency graph to detect cycles, i.e., consistency violations. To our knowledge, their approach is currently limited to storage systems offering at least Causal Consistency [16] which is not supported by most cloud storage services like our example Amazon S3 or NoSQL systems.

Patil et al. [32] also measure staleness in terms of time within their implementation of YCSB++ extending the original YCSB [6] benchmark. Their approach, though, can only measure rough approximations of actual consistency behavior due to the way values are measured.

Bailis et al. [33] use the distribution of network link latencies and quorum configurations to simulate consistency behavior for Dynamo-style [34] quorum systems. Their approach is, of course, limited to the usual bounds of simulation precision but offers a more cost-effective way for studying consistency behavior. It is unclear, though, whether Amazon S3 is actually a quorum system[12] and what network link latencies and a potential quorum configuration might look like. Hence, their approach cannot be used to study the consistency behavior of cloud storage services.

To our knowledge, no publications on consistency monitoring exist. All consistency benchmarking approaches might also be candidates for consistency monitoring but are currently too expensive to run continuously. We believe that all existing benchmarking approaches could be used as benchmarking component of our *Indirect Consistency Monitoring* method. The approach by Rahman et al. [13] could – constrained by the limitations outlined in section V-F – be an alternative to tracking KPIs and, thus, enhance our *Indirect Consistency Monitoring* method.

VII. Conclusion

In this work, we started with a short introduction to different consistency definitions, perspectives and dimensions. Afterwards, we recapped our approach for benchmarking consistency [8] and presented a set of extensions.

Next, we described the results of the long-term experimental study which we ran between August 2011 and September 2013 using Amazon S3 as example. This study showed a large variance in results over time: In 2011, we first saw daily and weekly patterns as well as sawtooth periodicity during the day. After we contacted Amazon in October 2011, these results worsened regarding average and mean values but the sawtooth pattern could no longer be seen as clearly as before. One year later, in October 2012, all periodicities were gone and S3 was effectively offering very good consistency levels. Until September 2013, these results had again deteriorated slightly – especially the variance of staleness values had increased.

Furthermore, when analyzing the probability of fresh reads as a function of the time since the last update, we noticed a curious "bump" in all our experiments which we believe is due to the master replica blocking briefly while forwarding updates to the second replica.

We also discussed that while MRC behavior was rather poor in 2011 and 2012, it is now very good with probabilities of violation well below 10^{-3}. RYWC and MWC were not violated in any of our experiments so that they at least seem to be guaranteed. Our recent addition of consistency benchmarking for delete operations showed similar behavior as the update operation and results were independent of the workload before issuing the delete.

Based on our experimental results and the more or less unexpected variance over time, we argue that cloud storage customers should continuously monitor the consistency behavior that their provider is delivering. For this purpose, we presented a new cost-efficient method for tracking changes in consistency behavior, *Indirect Consistency Monitoring*. The basic idea of our method is to monitor consistency behavior indirectly by tracking KPIs that are directly affected by changes in consistency behavior and to re-benchmark the system only then when application-visible changes have affected the KPIs. This way the frequency of expensive consistency benchmarking runs can be reduced. Additionally, choosing KPIs that are both easy and inexpensive to track asserts cost-efficiency of this method.

[11] mongodb.org

[12] In 2007, Vogels [35] mentioned that "Dynamo and similar Amazon technologies are used to power parts of our Amazon Web Services, such as S3." – for cloud storage services, of course, this may change anytime.

Depending on the results of *Indirect Consistency Monitoring*, appropriate measures can be taken like changes to the application implementation or business details, or even switching the provider. The latter is hindered by high degrees of vendor lock-in which current cloud customers incur due to a lack of standards. We believe that more efforts need to be put into standardization to maintain the customer's flexibility.

ACKNOWLEDGMENT

The authors would like to thank Amazon Web Services who provided research grants for our experiments. We would also like to thank Anton Tallafuss who benchmarked S3's consistency behavior for delete operations.

REFERENCES

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

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

[3] G. Young, "Quick thoughts on eventual consistency," *http://codebetter.com/gregyoung/2010/04/14/quick-thoughts-on-eventual-consistency (accessed Jun 21,2013)*, 2010.

[4] D. Terry, "The impact of eventual consistency on application developers," *http://littlemindslargeclouds.wordpress.com/2013/09/27/the-impact-of-eventual-consistency-on-application-developers/ (accessed Jan 13,2014)*, 2013.

[5] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai, "A middleware guaranteeing client-centric consistency on top of eventually consistent datastores," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2013.

[6] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[7] D. Bermbach, S. Sakr, and L. Zhao, "Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services," in *Proceedings of the Fifth TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2013)*. Springer, 2013.

[8] D. Bermbach and S. Tai, "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*. ACM, 2011, p. 1.

[9] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade offs in commercial cloud storages: the consumers' perspective," in *5th biennial Conference on Innovative Data Systems Research, CIDR*, vol. 11, 2011.

[10] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie, "What consistency does your key-value store actually provide," in *Proceedings of the Sixth Workshop on Hot Topics in System Dependability (HotDep)*, 2010.

[11] W. Golab, X. Li, and M. Shah, "Analyzing consistency properties for fun and profit," in *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2011, pp. 197–206.

[12] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "Ycsb++: benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 9.

[13] M. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. Wylie, "Toward a principled framework for benchmarking consistency," in *Proceedings of the 8th Workshop on Hot Topics in System Dependability*, 2012.

[14] P. Bailis, S. Venkataraman, M. Franklin, J. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *PVLDB*, vol. 5, no. 8, 2012.

[15] A. Popescu, "Consistency in the acid and cap perspectives," *http://nosql.mypopescu.com/post/4373459618/consistency-in-the-acid-and-cap-perspectives (accessed Aug 1,2013)*, 2011.

[16] D. Bermbach and J. Kuhlenkamp, "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.

[17] A. S. Tanenbaum and M. V. Steen, *Distributed Systems - Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 2007.

[18] W. Vogels, "Eventually consistent," *Queue*, vol. 6, pp. 14–19, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[19] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*. IEEE, 1994, pp. 140–149.

[20] R. Fischer, C. Janiesch, J. Strach, N. Bieber, W. Zink, and S. Tai, "Eine Bestandsaufnahme von Standardisierungspotentialen und -lücken im Cloud Computing," in *Proceedings of the 11. Internationale Tagung Wirtschaftsinformatik (WI)*, 2013, pp. 1359–1373.

[21] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze, "Cloud federation," in *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011, pp. 32–38.

[22] D. Bermbach, T. Kurze, and S. Tai, "Cloud federation: Effects of federated compute resources on quality of service and cost," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2013.

[23] D. Bermbach, M. Klems, S. Tai, and M. Menzel, "Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 452–459.

[24] K. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 187–198.

[25] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a case for cloud storage diversity," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 229–240.

[26] Broberg, Buyya, and Tari, "Creating a Cloud Storage Mashup for High Performance, Low Cost Content Delivery," in *Service-Oriented Computing–ICSOC 2008 Workshops*. Springer, 2009, pp. 178–183.

[27] J. Broberg, R. Buyya, and Z. Tari, "MetaCDN: Harnessing 'Storage Clouds' for high performance content delivery," *Journal of Network and Computer Applications*, vol. 32, no. 5, pp. 1012–1022, 2009.

[28] M. Klems, D. Bermbach, and R. Weinert, "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, Inproceedings, to appear.

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

[30] L. Lamport, "On interprocess communication," *Distributed computing*, vol. 1, no. 2, pp. 86–101, 1986.

[31] K. Zellag and B. Kemme, "How Consistent is your Cloud Application?" in *SoCC*, 2012.

[32] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "Ycsb++: benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 9.

[33] P. Bailis, S. Venkataraman, M. Franklin, J. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *PVLDB*, vol. 5, no. 8, 2012.

[34] 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 *Proc. SOSP*, 2007.

[35] W. Vogels, "Amazon's dynamo," *http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html (accessed Jan 13,2014)*, 2007.