Who Watches the Watchmen?
On the Lack of Validation in NoSQL Benchmarking

Wolfram Wingerath, Steffen Friedrich, Felix Gessert, Norbert Ritter

Databases and Information Systems Group
University of Hamburg
{wingerath, friedrich, gessert, ritter}@informatik.uni-hamburg.de

Abstract: There are numerous approaches towards quantifying the performance of NoSQL datastores with respect to dimensions that are notoriously hard to capture such as staleness or consistency in general. Many of these approaches, though, are built on assumptions regarding the underlying infrastructure or the test scenario and may lead to invalid results, if those assumptions do not hold. As a consequence, in-depth knowledge of both the system under test and the benchmarking procedure is required to prevent misleading results. In this paper, we want to make the case for more experimental validation in NoSQL benchmarking to uncover the bounds of existing benchmarking approaches.

1 Introduction

Relational databases (SQL databases) were once considered to be the one solution for all data storage and management problems. Basically, they all share the same query language and have very similar properties and performance characteristics. Everyone has a rather clear picture of what a relational database can do and what the implications of the different ACID flavours are (e.g. in terms of possible anomalies). Since many performance characteristics of SQL databases are dictated by their design, a reasonable performance evaluation of different SQL databases can be conducted by comparing their throughput and request latencies under a given workload for all supported ACID isolation levels. In contrast to the rather uniform SQL landscape, though, many different architectures with very different properties and performance characteristics are subsumed under the term **NoSQL**. In 2010, Yahoo! published the Yahoo! Cloud Serving Benchmark (YCSB) [CS+10] as the first noteworthy attempt at a benchmark across different NoSQL systems.[1] However, the YCSB only captures raw performance in terms of request throughput and latency under simple CRUD operation mixes (insert, read/scan, update, delete) and neglects all other criteria that describe the performance of distributed database systems. While measuring throughput and latency may have been sufficient to support a conclusive performance comparison between SQL databases, it is by far not enough to do the same in the field of NoSQL databases. During normal (i.e. fault-free) operation, throughput and latency are key to every NoSQL database, but the degree to which data are consistent (are there

[1] Several benchmarks had already been available before YCSB, but they were mostly built for performance verification throughout development and were not widely used to compare different systems.

stale reads? How stale are they?) and whether as well as in which scope transactions are available (single-key? Multi-key? Only across keys in the same data partitions?) cannot be ignored. Unlike traditional relational databases, many NoSQL systems are explicitly designed to withstand and even automatically recover from single-node outages, different kinds of network partitions or other failures that are guaranteed to happen in a distributed environment. Therefore, information on the impact that these anomalies have on consistency, durability, read and write availability or more generally request throughput and latency are crucial for evaluating NoSQL database system performance. In [FW+14], we already surveyed current efforts on the quantification of these performance characteristics (and staleness in particular) which are notoriously hard to capture and we concluded that many approaches rely on implicit assumptions that do not hold in practice, e.g. perfect clock synchronisation between different physical machines or low network communication delays. However, it is hard to evaluate the severity of such conceptual flaws from theory alone: just like a flawless benchmarking concept may lead to wrong measurements when implemented badly, a measurement scheme that allows potentially unbounded imprecision may yield useful results in practice.
In this paper, we argue that not only the concept behind a measurement scheme should be validated in order to explore the boundaries within which valid results can be obtained, but the actual implementation should be validated as well. In particular, we make the following contributions: we motivate practical benchmarking validation by illustrating weaknesses in existing NoSQL benchmarks (Section 2) and describe **SickStore** (Section 3), a single-node datastore we developed for validation that behaves like a distributed system and can simulate anomalies such as staleness. As a demonstration of SickStore's usefulness, we further conduct an experimental validation of staleness values obtained with YCSB++ (Section 4). Section 5 concludes the paper.

2 Why Validate Benchmarking Tools?

There are several approaches towards benchmarking staleness in distributed database systems that conduct their measurements in such a way that there are no guarantees on precision. In [FW+14], we described the ideas behind a number of database benchmarks and pointed out several weak spots in their design, concentrating on work related to staleness measurements. In this section, we first present our notion of staleness and then provide more detailed insights into the two staleness benchmarks that are most sophisticated in our opinion. Our goal is to convince the reader that benchmark validation is important as there are various possible sources of arbitrary imprecision, for example clock drift, network latency or mere software faults.
What is Staleness, Anyway? A stale read is a read operation that returns an outdated object version. When a data object is written to an asynchronously replicated datastore, there is a certain timeframe during which the different replicas of this data item diverge, because the write is visible on some, but not all replicas. This timeframe represents the actual data-centric staleness window, because stale reads of the written object may be served as long as the write operation has not been applied to all replicas. Data-centric staleness cannot be measured easily, though, because it requires knowledge of the internal system state and therefore cannot be implemented in a system-agnostic and generic way. Furthermore, the

staleness window that is actually observable on the client-side during a given workload might be sufficient for many use cases or even more relevant than the exact point in time at which data in the system become consistent. Therefore, virtually all existing staleness benchmarks measure the client-centric staleness, i.e. they try to capture how long after a write operation the system keeps serving stale data.

Bermbach et al. Staleness Measurement. Bermbach et al. [BT14] create a global log file from multiple distributed servers under the assumption of perfect clock synchronisation. As illustrated in Figure 1, the basic setup comprises the datastore under test, one server that runs a common YCSB workload and one writer and several readers for the staleness measurement.[2] To determine the staleness window of a write operation, the writer periodically writes its local time and a version number to the datastore, while the readers repeatedly poll the written data item and log their own local time for each received value. The staleness window can eventually be computed as the difference between the timestamp of the last stale read (local time of a reader) and the timestamp of the initial write (local time of the writer). The timestamps that are used for the computation of the staleness window are highlighted in the illustration: the writer updates the version number at timestamp 3 and the two readers do not see the updated version before timestamps 8 (Reader 1) and 7 (Reader 2), so that the staleness window is computed as $8 - 3 = 5$. Obviously, this procedure only delivers an approximation of the staleness window whose precision depends on how well the local clocks of the writer and the readers are synchronised. For example, only the last read timestamp of Reader 1 is used for the computation, since only the *last* stale read is relevant which (under the assumption of synchronised local clocks) comes from Reader 1. Accordingly, if the writer and Reader 1 are perfectly in sync, while Reader 2 lags 10 time units behind, only the last stale read from Reader 2 taken into account and the measured staleness window becomes $16 - 3 = 13$. If the writer's local clock lags behind or if one of the reader's local clock is significantly faster, it may even become negative. Clock drift has already been identified as a possible cause for dubious experimental results

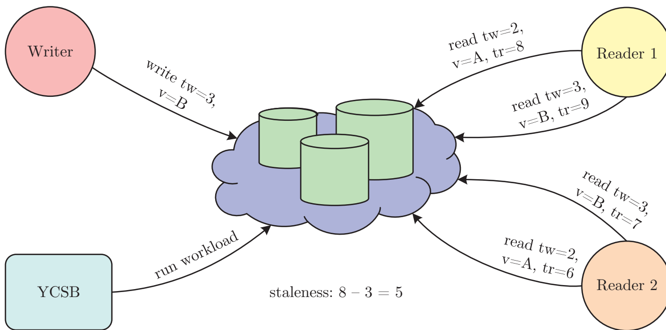


Figure 1: The basic set up of the staleness benchmark by Bermbach et al. [BT14].

by Bermbach et al. themselves [BZS14]. But even in a well-synchronised environment, this approach still does not produce exact results, because the timestamp that is written

[2]Writer and reader are placed on separate machines for several reasons: first, they might experience reduced or no staleness at all when placed on the same machine, e.g. when their requests are routed to the same replicas. Second, a single machine might not be able to saturate the system under test. Third, real-life clients are also (geographically) distributed.

to the datastore represents the point in time directly *before the write request was sent*, whereas the point in time *directly after the write request was acknowledged* is when data actually become stale. Depending on the required precision, the network latency and the time the system needs to process the write request, the staleness measurement may or may not be a reasonable approximation of the actual staleness window. Apart from that, the read/write ratio of the YCSB workload may be significantly disrupted by the additional reads and writes during the staleness measurement.

YCSB++. Another example of a staleness benchmark that gives rise to potentially unbounded measurement error[3] is the YCSB++ [PP$^+$11] which, in contrast to the approach by Bermbach et al., does not rely on clock synchronisation, but instead works with notification between writer and reader[4]. In principle, the writer inserts[5] new objects and enqueues their keys to a queue administered by a ZooKeeper service on a dedicated server, while the reader repeatedly dequeues, tries to read the corresponding values and eventually approximates the staleness window as the lag between the initial receipt of a key and the retrieval of the inserted value. Figure 2 shows the individual steps of a YCSB++ staleness measurement with one writer and one reader in a simplified way. The individual steps of the experiment are demarcated as i_n where i represents *the order* in which all steps happen and n represents *where* (at which node) each step happens (r for reader, w for writer, s for store). For illustration purposes, we use these demarcations as though they were (global) timestamps if appropriate. Further, we assume write operations to be applied at the moment of acknowledgement and we assume read operations to take place instantaneously when they are received. Initially, both writer and reader are given the signal to start the experiment by ZooKeeper (received at 1_w and 1_r, respectively). The reader tries to dequeue (2_r) the first element of the ZooKeeper queue, but has to wait, since the queue is empty. Meanwhile, the writer issues (3_w) the request to insert object x. The datastore receives the request (4_s), processes it and finally sends (5_s) the acknowledgement back to the writer. When the writer has received the acknowledgement (6_w) and enqueues (7_w) x to the ZooKeeper queue. The reader is notified of the new key in the queue, dequeues it and receives it (8_r). Object x. According to the basic idea as we also described it in our survey [FW$^+$14], the reader repeatedly requests object x, until x is actually returned. In more detail, though, the reader communicates with the ZooKeeper service after every unsuccessful read attempt: After the first read request (issued at 9_r, received at 10_s) returns null (sent back at 11_s, received at 12_r), the reader pushes x back to the ZooKeeper queue, removes the next item in the queue and continues the procedure with this item. When the reader eventually removes x again from the queue (13_r), it starts another read attempt (issued at 14_r, received at 15_s, answered at 16_s) and finally receives (17_r) the inserted object x. The staleness window is then approximated it as $17_r - 8_r$, i.e. the time between the reader receiving a key and the reader successfully retrieving the corresponding value. The *data-centric staleness window* starts at 5_s and ends somewhere after 10_s and before 15_s. The *client-centric staleness window* is $10_s - 5_s$, but it cannot be computed, because

[3] Our initial understanding of the YCSB++ approach was that it provided a lower bound for the staleness window as we reported in our survey on NoSQL OLTP benchmarking [FW$^+$14]. We have to admit, though, that that this was not the case and that YCSB++ does not provide any bounds on precision during staleness measurements.

[4] In the YCSB++ paper, writer and reader are referred to as **producer and consumer**.

[5] Note that staleness measurement are only possible for inserts and *not* for updates.

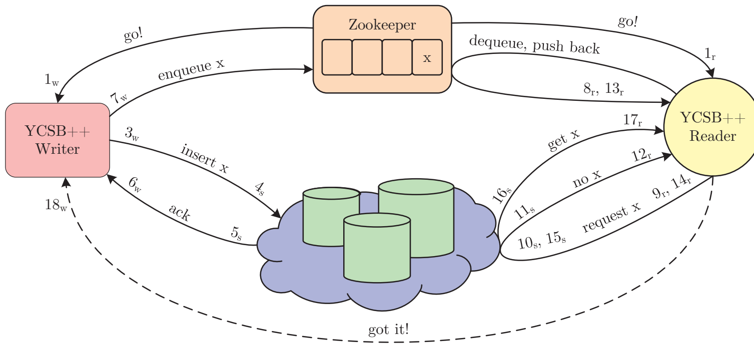


Figure 2: The basic experimental setup for staleness measurement in YCSB++.

all timestamps are taken from client machines and therefore neither 10_s nor 5_s are available. Further, the difference between 10_s and 5_s would also be distorted by clock drift, even if they were available, because the insert and the read request are not necessarily processed on the same datastore node: if the exact same machine that acknowledged the insert also responded to the read request, most systems would not exhibit any staleness whatsoever, since x would be visible directly after the acknowledgement. Hence, in case of an actual stale read, the insert and read request may be assumed to have been processed by two distinct machines that do not share the same local clock. But even though the exact client-centric staleness window cannot be obtained, it is possible to completely rule out the possibility of clock drift by using only timestamps from either the writer or the reader to approximate the staleness window: since $5_s < 8_r < 9_r < 10_s$, a *lower bound* for the staleness window can be computed as $9_r - 8_r$.[6] Similarly, an *upper bound* could be computed, if the reader reported back to the writer after having read x. Assuming the writer received such a notification from the reader at timestamp 18_w, an upper bound could be computed as $18_w - 3_w$, since $3_w < 5_s < 10_s < 18_w$ holds. Since the staleness window measured by YCSB++ does not contain the actual staleness window ($10_s - 5_s$) and is not contained in it, it is neither a lower nor an upper bound: depending on the network delay and the time that an item remains in the ZooKeeper queue, there may be significant lag between the writer enqueueing the key and the reader initially receiving it (**read-after-write lag**) as well as between the reader pushing the key back into the queue after an unsuccessful read attempt and receiving it again. The former kind of lag may prevent the reader from observing any staleness, simply because it is not notified soon enough, while the latter may lead to arbitrarily large staleness measurements, because the reader takes too long to start another read attempt. YCSB++ avoids imprecision through clock drift, but allows additional delays through network commutation and the use of a queue which might not be delay between the reader and the ZooKeeper coordination service. In addition and similar to the experiments by Bermbach et al., it is unclear in what way the staleness measurements influence the read/write ratio of the YCSB workload that is run by the writer.

[6]Note that this lower bound can be very close to zero, if the read is successful at first or second attempt. If more than two read attempts are required, though, the lower bound can assume values significantly greater than zero.

3 Designing for Single-Node Inconsistency: SickStore

The major difficulty in validating existing staleness benchmarks is that none of the time-stamps that would be required to determine server-side (data-centric) staleness are actually available for measurement, either because they would require internal system knowledge or because the database system itself is distributed and therefore subject to clock drift. In order to have a gold standard that can be used to evaluate the quality of staleness benchmarking results, we developed a **s**ingle-node **inc**onsistent **k**ey-value **store**: **SickStore**. SickStore is a single-node key-value store that is able to simulate the behaviour of a distributed key-value store. It is thus able to provide the same anomalies as a distributed system, but at the same time consistent knowledge about the system state at any point in time. Its main building blocks are the internal backend that holds all data, several virtual server nodes that listen on different ports of the same machine and a processing layer in between that does the actual request processing and decides what data can be served from what server node. Figure 3 shows an example where one write request and two read re-

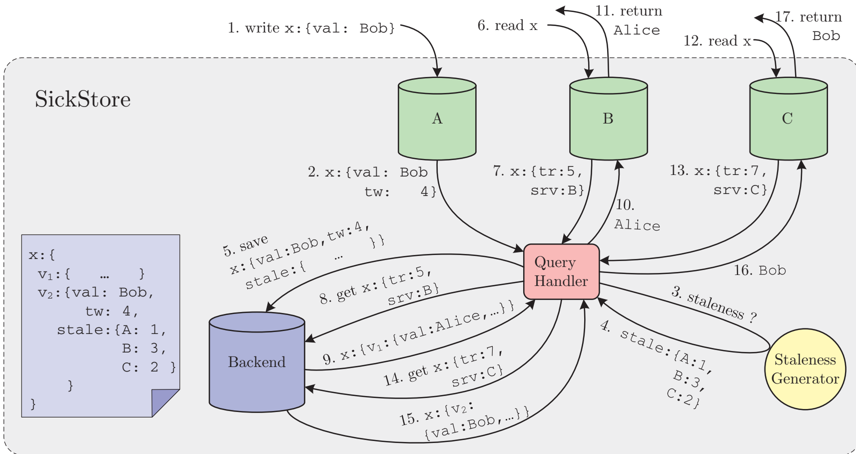


Figure 3: The basic architecture of SickStore: the different virtual server nodes accept and reply to all requests, but they all display a certain staleness, while the central versioned key-value store (backend) keeps track of all changes and is aware of the true system state at any time.

quests are executed against a SickStore instance with three virtual servers A, B and C. A write request to insert or update value “Bob” under key x arrives at timestamp 4 at server A (1) which forwards the request including the key, the value and the write timestamp $tw = 4$ to the query handler (2). The query handler requests the staleness generator to generate a staleness window for each virtual server (3) and receives one in form of a map that associates each server with its individual staleness window in ms. The query handler has the backend store everything (value, write timestamp, staleness associations) under key x (5). The backend maintains a list of all writes that have occurred under every key in order of their arrival at the backend. The acknowledgement goes back to the server that received the request and is then returned to the client.

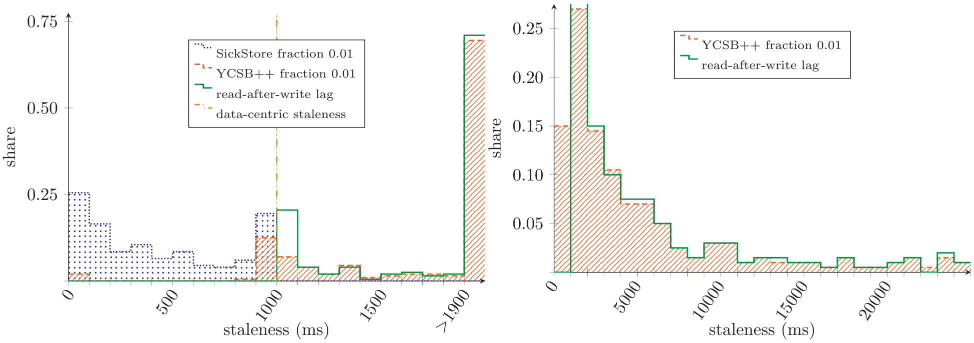
Shortly thereafter at timestamp 5, a read request for x arrives at server B (6). Similar to the request discussed above, this read request is also forwarded to the query handler together with the read timestamp $tr = 5$ and the name of the server node that received the request (7). The query handler then directly sends these information to the backend (8) and requests the most recent version of x that server B is allowed to serve; since the most recent version has been written at timestamp 4, but has a staleness window of 3 associated with server B, it is not visible for server B before timestamp $4 + 3 = 7$. Since the current read timestamp is 5, the query handler receives (9) the value of the most recent version that *actually is* visible for server B, in this example the value "Alice", and hands it to server B (10) which then returns it to the client (11). Another read request for the same key arrives at server C at timestamp 7 (12) and is again forwarded to the query handler (13) together with the read timestamp $tr = 7$ and the name of the server C. Again, the query handler requests the most recent visible version, but actually receives the most recent version this time, because the read request arrived at timestamp 7 and the C-associated staleness window is 2 (i.e. because $7 \geq 4 + 2$). Naturally, the value is eventually returned to the client. Since all SickStore components have the same local clock time and since there is virtually no communication delay between them, the processing layer can precisely implement a pre-defined visibility delay for each virtual server and thus make sure the data are served exactly as stale as they are supposed to be served. Further, the modular design of SickStore allows to interchange different implementations of the staleness generator, so that our constant staleness generator can easily be replaced by a more sophisticated one in the future. Depending on what criteria are used to determine the staleness that is exhibited, arbitrary replication or sharding strategies in arbitrary network topologies can thus be simulated.

4 Experimentally Validating Latency Measurements With YCSB++

The benchmarks discussed in Section 2 (and presumably many more that we do not discuss here) bear the potential to display arbitrary measurement error without the user noticing. As a consequence, their precision and actual meaningfulness are hard to evaluate by merely speculating about the impact of workload disruption, clock drift (Bermbach et al.), communication delay (YCSB++) or other possible ramifications in a given experimental setup. We claim that a performance evaluation using these benchmarks is not possible without experimental evidence regarding whether, under what circumstances and to what extent these benchmarks are actually useful. In [BZS14, p. 42], Bermbach et al. already concluded that clock drift during their measurements has severe implications on the measured staleness and thus can easily lead to misleading results. In this section, we show that YCSB++ is also prone to severe measurement errors, even though clock drift causes none of them.

Experimental Setup. All experiments were executed in a virtualised environment hosted on Xeon servers with 2.1 GHz 6-core CPUs: the YCSB++ writer and reader each had two VCPUs and 4 GB of memory, the ZooKeeper service and SickStore each had a single VCPU and 2 GB and 30 GB of memory, respectively. In principle, the YCSB writer is capable of executing an ordinary YCSB workload against the datastore under test and submits only a small *fraction* of the insert operations (1% by default) to ZooKeeper for staleness measurements. To keep the fraction scalable in the range between 0% and 100%,

we decided to have the YCSB++ writer run an insert-only workload and thus avoid any operations whose staleness could not be measured.[7] To make sure that neither SickStore nor the YCSB++ writer would not become bottlenecks during the experiments, we determined their maximal throughput, first: our SickStore deployment achieved consistently more than 3000 operations per second under insert-only, read-only and mixed insert-read (50/50) workloads using single-threaded vanilla YCSB and also under an insert-only workload using a single-threaded YCSB++ writer with staleness measurement disabled[8]. To further prevent any dependencies between individual experiments, we shutdown SickStore's JVM after each experiment and started a new instance for each experiment. We configured SickStore to exhibit *constant staleness of exactly 1000 ms*.



(a) The YCSB++ staleness measurements do not reflect the actual data-centric well.

(b) The measured staleness distribution has a long tail and many outliers.

Figure 4: Target throughput 500 ops/sec, fraction 1%, 1000 ms data-centric staleness.

Measuring Staleness. We started our first experiment with a fraction of 1% as proposed in the YCSB++ paper and we only used a target throughput of 500 inserts per second which translated to 5 inserts per second with subsequent staleness measurement. Both writer and reader were single-threaded the experiment stopped after 200 staleness measurement. The histogram in Subfigure 4b illustrates the frequency of actual client-observable staleness (dotted line), the staleness windows measured by YCSB++ (dashed line), the constant data-centric staleness (dotted line) and the read-after-write lag measured at SickStore (solid line), i.e. the time between the initial write and the last (successful) read operation. Since YCSB++ tries to measure the actual data-centric staleness as an approximation of the read-after-write lag and since the actual data-centric staleness window exhibited by SickStore was exactly 1000 ms, one would expect the measured staleness windows and also the read-after-write lag to be close to 1.000 ms. As illustrated in the plot, though, the experimental results did not correspond to our expectations at all. First and foremost, less than 25% of all measured staleness values (47 out of 200) are between 900 and 1000 ms, 4 operations were not observed as stale at all and more than half of all operations (119) displayed 1900 ms of staleness or more.

The histogram of all values in Subfigure 4b shows that some staleness values reported by

[7] As described in Section 2, YCSB++ Only measures staleness values for inserts and not for update operations.

[8] We set the fraction of staleness measurements to 0, but also started a single-threaded consumer and a ZooKeeper and had them perform their usual barrier synchronisation.

YCSB++ exceed the actually exhibited staleness by more than the 20-fold. This behaviour can be explained with the frequency distribution of the read-after-write lag which appears to dominate most staleness measurements. Since the read-after-write lag represents the time that passes between the initial write and successful read, it also indicates the time that passes between the last unsuccessful read and the final successful read, because any read that is received by SickStore more than 1000 ms after the initial write would automatically result in a success and thus would finalise the read-after-write lag for that value. Therefore, the time that lies between the last two read operations of a measurement can at most be 1000 ms smaller than the read-after-write lag: if the first read occurs directly after the initial write, the read-after-write lag and the measured staleness are almost equal, whereas a delay between the initial write and the first read that is greater than the data-centric staleness window would result in a zero-staleness measurement, because the first read would be successful. Since only few zero-staleness measurements occurred in our first experiment, the great number of large read-after-write lag values can only mean that there were many staleness measurements where the first read was executed within the first second after the initial write (resulting in an unsuccessful read), so that the reader had to push back the corresponding key to ZooKeeper, but then did not receive it again for a substantial amount of time (tens of seconds in some cases).
Our logs revealed that the YCSB++ reader actually achieved more than 320 reads per second, irrespective of the configured fraction, but also that it read the exact same keys dozens of times in succession instead of cycling through them as expected. Reviewing the YCSB++ code, we found out that the reader does not actually dequeue items from ZooKeeper, but instead uses the first item from the list that is returned by calling the `getChildren` method; according to our experiments (and the ZooKeeper documentation for various releases), though, this procedure does not guarantee the keys to be returned in any specific order which explains why the reader repeatedly dequeues the exact same keys. We verified through further experiments that the usefulness deteriorates even further when the fraction is increased: with the fraction set to 50%, virtually all staleness values are either way too large or zero. We managed to achieve a reasonable staleness approximation with YCSB++ at 1 staleness measurement per second (see Figure 5). However, this setting is not reliable: in an experiment with an increased data-centric staleness of 5000 ms demonstrate, the measured values were wildly scattered again, peaking at over 180 s.

Summary Using SickStore, we were able to uncover substantial imprecision in an experimental setup that would otherwise have been hard to detect. In particular, we could establish that the YCSB++ staleness measurement corresponds to neither the actual data-centric staleness nor to the real distribution of the staleness that is perceived by the reader. Further, we found out that YCSB++ provides barely any control over the degree of workload disruption, even though this is advertised in the YCSB++ paper: the reader always runs

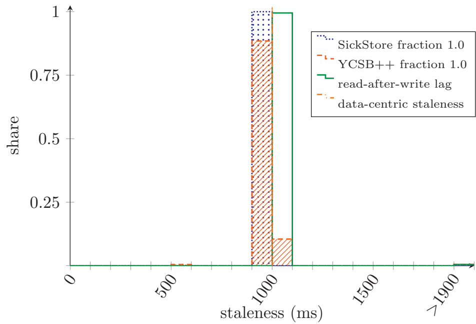


Figure 5: Target 1 ops/sec, fraction 100%: it is possible to achieve reasonable results.

unthrottled, irrespective of the configured fraction. The most severe problem for staleness measurement is that the YCSB++ reader mostly enqueues and dequeues the exact same keys in direct succession, so that some keys are not requested at all for tens of seconds which leads to significant distortions. Depending on the number of keys stored at ZooKeeper as well as the actually exhibited staleness, the resulting overhead may completely dominate the experimental results as it can be orders of magnitudes greater than the value that should actually be measured.

5 Conclusion and Open Challenges

The great variety of feature sets that are available on the datastore market also brings confusion with respect to what guarantees a datastore can actually provide, particularly with respect to performance characteristics that are hard to quantify objectively. To enable an evaluation of the different benchmarks that try to address this issue, we presented our prototypical database for validation, **SickStore**, that mimics the behaviour of a distributed a database system and simulates anomalies, but runs on a single server, so that all events can be associated to global timestamps without any clock drift. We described SickStore's architecture and its main features. In our experimental validation, we demonstrated how SickStore can be used to doublecheck staleness measurement results by comparing SickStore's internal logs to the YCSB++ measurements. Thus, we uncovered that YCSB++ is likely to produce misleading staleness values without the user noticing.
In order to enable validation of other measurements apart from staleness benchmarks, we are currently implementing freely configurable transactional anomalies as well as request latency and read/write throughput into SickStore. Further, we are building more sophisticated staleness generators that simulate different replication or sharding strategies and allow a more fine-grained tuning of staleness between the virtual SickStore server nodes.

References

[BT14] David Bermbach and Stefan Tai. Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies. In *Proceedings of the 2nd IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2014. Best Paper Runner Up Award.

[BZS14] David Bermbach, Liang Zhao, and Sherif Sakr. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In *Performance Characterization and Benchmarking*, volume 8391 of *Lecture Notes in Computer Science*, pages 32–47. Springer International Publishing, 2014.

[CS+10] Brian F. Cooper, Adam Silberstein, et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154. ACM, 2010.

[FW+14] Steffen Friedrich, Wolfram Wingerath, et al. NoSQL OLTP Benchmarking: A Survey. In *Informatik 2014*, volume 232 of *Lecture Notes in Informatics (LNI)*, pages 693–704. Gesellschaft für Informatik (GI), 2014.

[PP+11] Swapnil Patil, Milo Polte, et al. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14. ACM, 2011.