NoSQL OLTP Benchmarking: A Survey

Steffen Friedrich, Wolfram Wingerath, Felix Gessert, Norbert Ritter

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

Abstract: In recent years, various distributed NoSQL datastores have been developed that offer horizontal scalability and higher availability than traditional relational databases, but fewer querying options and only reduced consistency guarantees. The diversity of the design space makes it difficult to understand the performance implications of individual system designs. Existing benchmarking tools measure some relevant aspects, but do not capture all of them. In this paper, we give an overview of the state-of-the-art in NoSQL OLTP benchmarking, identify missing features as well as open challenges and point towards possible solutions.

1 Introduction

Traditional relational database management systems (*RDBMSs*) were designed to be the one solution to all data management and storage problems and thus they all share a broad foundation: They provide the full SQL spectrum (including joins) and are not highly available as they follow strongly consistent ACID semantics. Hence, these systems can be compared reasonably well by benchmarks (e.g. TPC) that quantify their performance solely along dimensions like throughput and request latency under realistic OLTP workload patterns.

During the last decade, however, a myriad of distributed NoSQL datastores have emerged that promise to fill the growing gap between what highly distributed (web) applications require and what traditional RDBMSs can provide by sacrificing consistency guarantees and querying options (hence the name) in favour of high availability and scalability. The design space is vast and the dependencies between desirable properties are complex. Even though high-level abstractions such as the CAP Theorem [Bre00, GL02] or the PACELC model [Aba12] can help to understand the basic trade-offs that are involved, the performance implications of individual system designs are often not obvious, so that the optimal data storage solution for an application cannot be chosen by simply comparing candidate specifications. Representative benchmarking results are not available, though, as current approaches do not capture all the relevant aspects and employ operation mixes as workloads that only loosely correspond to real-world scenarios.

In this paper, we survey the in our opinion most relevant NoSQL OLTP benchmarks, point out missing features, discuss open challenges and possible ways to approach them.

In Section 2, we survey current efforts in evaluating NoSQL database systems according to request latency and throughput, availability and consistency in its various forms. We briefly present a novel approach to benchmarking consistency in Section 3 and give a summary of open challenges and a conclusion in Section 4.

2 Comparing Distributed Datastores

Data in distributed datastores are spread (sharded) across several nodes (servers). In order to prevent data loss and to preserve availability in the presence of errors, data are also replicated across several nodes. But sharding and replication have severe implications on performance: While replication can increase read performance as the same data items are available on several machines, it can also increase write latency (synchronous replication) or give rise to stale reads or conflicting writes (asynchronous replication). Furthermore, sharding introduces an overhead to operations that span data items on different nodes; operations such as joins over datasets on different nodes become infeasible and global invariants are hard to maintain. As no single system can achieve all desirable properties at once, each NoSQL datastore is optimised for a specific property set.

In the remainder of this section, we discuss different dimensions along which NoSQL datastores can be evaluated and compared. First, we cover the most basic form of benchmarking that translates to measuring raw performance in terms of request latency and operational throughput, concentrating on YCSB as the most popular and widely accepted benchmarking framework in the field of NoSQL OLTP benchmarking. We then consider related work on the quantification of availability. Finally, we discuss different notions of consistency in distributed databases and examine approaches towards their measurement.

2.1 Request Latency and Operational Throughput

The de facto standard for NoSQL OLTP benchmarking is the *Yahoo Cloud Serving Benchmark (YCSB)* framework which was published in 2010 [CST$^+$10]. Primarily targeting key-value store functionality, YCSB is a highly generic general-purpose OLTP benchmark that can easily be extended to support additional databases as it only requires the implementation of a simple CRUD interface (read, scan, insert, update, delete). In contrast to other benchmarks such as TPC-C that evaluate system performance under realistic workloads, YCSB is built around the idea of examining different characteristics of a datastore using distinct architectural *tiers*: A tested database is stressed with CRUD operations by the YCSB Client to measure throughput, operations per second and request latency under high load (*performance tier*). YCSB workloads do not model real-world applications, but are customisable mixes of read/write operations that represent entire application classes: Random distributions[1] determine which operations are performed, which records are read/written, how frequently they are read/written and other workload details. The records that are read and written consist of a number of (random) ASCII string attributes; the number

[1] YCSB supports uniform, multinomial and variants of Zipfian distributions.

of columns and the size of each value can be configured. Scalability and elasticity can be quantified by measuring the increase in performance as machines are added to the system (*scalability tier*).
The generic workloads and the simple CRUD interface make YCSB suitable to explore general trade-offs in a wide class of storage systems. But due to this simplicity, YCSB does not account for functionality beyond that of simple key-value stores and abstracts from many differences such as data models. Seeing that many NoSQL systems actually are highly specialised for particular distributed large-scale applications, a performance evaluation for specific real-world tasks appears desirable. Furthermore, the single-node client poses a scalability problem for benchmarking in large-scale environments as it cannot saturate large distributed database systems under test. Additional tiers for characteristics like availability, replication or fault tolerance were proposed in the original contribution, but not implemented in YCSB.

BG [BG13] is another benchmarking framework which, in contrast to YCSB, is still actively developed. It models actions in a social networking application and therefore provides a richer conceptual model and requires the implementation of a more complex interface than YCSB for operations such as listing all friends of a given user or returning the top-k posts on a user's profile. In addition to throughput and latency measurements as in YCSB, BG supports SLA (service level agreement) conformance checks: The *Social Action Rating (SoAR)* represents the highest throughput that can be sustained without violating a given SLA.

2.2 Availability

A partition-tolerant available system can always accept read and write requests by clients and will eventually return a meaningful response, i.e. not an error message. However, anomalies such as network partitions or server crashes occur on a daily basis in large distributed environments and therefore data have to be replicated across different failure domains, so that operation can be sustained in spite of such anomalies. Since availability is of paramount importance for many applications, information on the performance impact of different replication strategies during normal operation or in failure scenarios are valuable.

The *Under Pressure Benchmark (UPB)* [FMdA$^+$13] aims at quantifying and comparing the availability of different distributed database systems by measuring the operational throughput under three different configurations: with replication turned off and no failing nodes, with replication turned on and no failing nodes and with replication turned on in the face of node failures. In every setting, the system under test is given some time to stabilise (warming period) before measurements are made. In order to generate a heavier workload than is possible with a single client, UPB uses several independent YCSB clients in parallel and aggregates the results.
While it does manage to quantify the impact of replication on steady-state performance during normal operation and after node failures, the UPB does not quantify availability well, in our opinion, because the measurements do not reflect performance problems *directly after* node failure.

In a 2013 evaluation, Engber et al. [NE13, Eng13] measure the *immediate* impact of node failures on operational throughput and therefore availability. Similarly to the UPB, they aggregate the results of several independent YCSB clients after a short warming period. As they induce node failure *during* measurement, the authors are able to observe real-time behaviour such as downtime during failover, performance drops caused by load balancing and, more generally, database performance before the system stabilises again.

2.3 Consistency

The CAP Theorem advertises that strong consistency is unachievable for a highly available system in the face of a partition, the underlying notion of strong consistency being *linearisability* which translates to the guarantee that reads and writes are always executed atomically and are sequentially consistent (linearisable [HW90]). In simple words, strong consistency guarantees that all clients have the same view on the data at all times. Some systems sacrifice availability to remain strongly consistent while others employ available, but only eventually consistent models: In an *eventually consistent (EC)* system, all replicas of a data item are guaranteed to converge in the absence of updates and partitions, i.e. they will reach an identical state at some point in the future. The order in which individual writes are applied, though, is arbitrary and dependent writes may become visible out of order. However, as demonstrated by current research [LFK$^+$11, BGH$^+$13], it is possible to simultaneously achieve EC and *causal consistency (CC)* at the cost of increased staleness and additional complexity in comparison to plain EC. Informally, any two operations on a data item in a CC system are executed in the order they are received, if one of them (directly or transitively) causally depends on the other.[2] EC and CC are *data-centric consistency* models that have their focus on the internal state of the storage system and synchronisation between replicas, whereas inconsistencies that can be observed by clients are captured by *client-centric consistency* models: *Monotonic Read Consistency (MRC)* mandates that a client that has observed version n of a data item will never observe this particular data item in a version less than n. *Monotonic Writes Consistency (MWC)* requires that two updates issued by the same client are executed in the order that they arrive at the storage system and *Write Follows Read Consistency (WFRC)*[3] guarantees that a write operation following a read of object x in version n is never applied to replicas with versions of x less than n. Under *Read Your Writes Consistency (RYWC)*, a client that has written version n of a data item will never observe this particular data item in a version less than n.

While strongly consistent systems display no inconsistencies with respect to neither ordering nor staleness and therefore provide all of the above-mentioned client-centric guarantees, many EC systems provide none[4]. Thus, it is hard to determine whether the advantages of using an EC datastore outweigh the disadvantages of potentially stale or conflicting data

[2] An operation o_1 is directly causally dependent on another operation o_2, if (1) the storage system receives o_1 before o_2 and both operations are issued by the same client or if (2) o_1 is an update and o_2 is a read that returns the result of o_1.

[3] Example: WFRC guarantees that a reply to a posting can never be observed before the initial posting.

[4] MRC, MWC and WFRC can be provided in an eventually consistent system.

in a particular application scenario.

In the following, we discuss ways to compare different distributed databases with respect to the ordering and staleness properties they exhibit. We also cover the notion of *transactional consistency* known from ACID databases which differs from distributed replica consistency and efforts in verifying consistency after system partitions.

2.3.1 Staleness

Given any highly available EC system, it is impossible to provide strict bounds on how strongly the different replicas diverge or how long it will take them to reach agreement once they are out of sync. However, various efforts have been made to predict or measure staleness for different systems.

With the *Probabilistically Bounded Staleness (PBS)* prediction model, Bailis et al. [BVF$^+$12] estimate the expected bounds on staleness in Dynamo-style datastores with respect to both versions and wall clock time on the basis of write propagation and read messaging delays. They introduce the two metrics *t-visibility* for the probability of observing a write t time units after it returned and *k-staleness* for the probability of reading one of the last k versions of a data item and then further combine them both in $\langle k, t \rangle$-*staleness consistency* to encapsulate the probability of reading one of the k latest versions of a value, given the latest value was written at least t time units ago.
On the downside, PBS does not treat the system as a black box, but instead requires internal knowledge of the storage system (delays) and abstracts from implementation details; as anti-entropy protocols such as read-repair or the use of Merkle trees are not taken into account, actually observed staleness may be less severe than predicted by PBS. Furthermore, it is only applicable to a specific class of distributed (Dynamo-style) databases.

In [WFZ$^+$11], Wada et al. describe an experimental setup to measure time-based staleness in different cloud databases. The authors describe different experimental configurations that employ one reader and one writer which are hosted on the same virtual machine (VM), on different VMs in the same datacenter and on VMs in different datacenters: The writer periodically writes its local time to a particular data item and the reader repeatedly retrieves this item; on discovery of a new timestamp, the reader computes the observed staleness window as the difference between its own local time and the observed timestamp.
As the authors do not mention whether or how the local clocks of writer and reader are synchronised, we assume there to be no synchronisation. Under this assumption, all experiments where writer and reader do not share the same machine are profoundly flawed, because they might contain arbitrary measurement errors. The significance of the remaining experiments appears limited as well since writer and reader might experience reduced or no staleness at all, e.g. because they are routed to the same replicas.

To address this issue, Bermbach et al. [BT11, BT14] propose an extension to the approach of Wada et al. that employs multiple readers: One writer periodically writes the current local timestamp and a version number, while each of several readers repeatedly polls the storage system and logs its local read timestamp, the observed write timestamp and the observed version number. In a subsequent analysis of all reader logs, t-visibility and k-

staleness are approximated under the assumption of perfect synchronisation of all client clocks. In addition to their consistency measurement component, they also run a YCSB workload with one YCSB client to saturate the storage system.
Similar to Wada et al., Bermbach et al. also rely on clock synchronisation through the cloud provider and thus significantly diminish the reliability of their results: As they assess themselves [BZS14], using local timestamps from different readers with potentially drifting clocks may lead to dubious results.

In contrast to the studies discussed above, Golab and Rahman et al. [RGA$^+$12, GLS11] from HP Labs aim to measure the observed consistency under a given workload instead of designing a workload to derive consistency properties. They criticise that other measurement techniques introduce artificial operations (repeated reads) that tend to disrupt the workload, stress the system under test considerably and thus may distort the results. They formally define Δ-atomicity as a time-based consistency property that informally requires a read operation to return a value that is at most Δ time units stale. In other words, a system provides Δ-atomicity, if every value becomes visible during the first Δ time units after[5] the acknowledgement of its write. Figure 1 illustrates the idea of how to compute the value Δ for a given database history: Several read and write operations are executed on data item x over time where the entirety of all operations regarding version i of x is referred to as zone Z_i. For each pair of zones that belong to the same data item and overlap in terms of time, a value χ is computed that corresponds to the width of the respective staleness windows. Δ is the maximum of all χ values, i.e. the largest observed staleness window. The goal of the authors' experiments is to quantify the consistency that is provided by a distributed datastore in terms of Δ-atomicity. The required information are collected by extended YCSB clients that log timing information for each operation.
Even though the workload design differs from the approaches discussed before, the actual measurements are very similar. While the precision of the described approach also depends on clock synchronisation, the authors state the error margin to be "around 1ms".

As part of the SLA conformance check, *BG* also quantifies the amount of stale and otherwise inconsistent data that are observed during experiments. A BG workload is executed by several distributed clients each of which operates on a logical partition of the dataset. Every client is aware of the initial state of the data and each update operation and therefore stale reads can be detected through log analysis.
While the different BG clients do not interact and therefore do not require clock synchronisation and do not suffer from communication delays between reader and writer, staleness and other anomalies may not be observable in some scenarios, as both read and writer share the same physical machine. Given that reader and writer in real-world social networking applications are often globally distributed, this appears as a major drawback in terms of realism.

An approach that does not rely on clock synchronisation is implemented in **YCSB++** [PPR$^+$11] where multiple clients are coordinated via one central ZooKeeper server [HKJ$^+$10]. Most notably, YCSB++ measures time-based staleness with multiple clients working together in a producer-consumer pattern as illustrated in Figure 2: Initially, the consumer client subscribes (1.) to be notified by ZooKeeper as soon as a specific object is

[5] For $\Delta > 0$, Δ-atomicity is strictly weaker than atomicity which demands immediate visibility for any write.

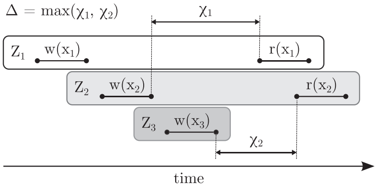

Figure 1: Example computation of Δ as the maximum of all χ values similar to [RGA+12, Figure 1]. Note that there is no χ value between Z_1 and Z_3 as the largest staleness window regarding x_1 is determined by χ_1.

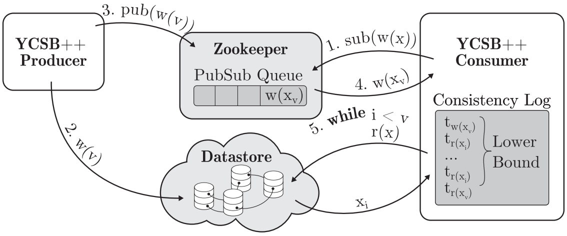

Figure 2: YCSB++ benchmark coordination and consistency measurement with Zookeeper as described in [PPR+11] .

.

updated by the producer. After the producer client has inserted or updated (2.) a record, it publishes (3.) the write to the ZooKeeper PubSub queue. The consumer client is notified (4.) via ZooKeeper and then repeatedly requests (5.) the updated record, until the new version is returned.

To provide a lower bound for the actual inconsistency window, the delay between the first attempt and the first successful attempt to read a record is measured by the consumer. To keep the impact of client coordination on the experimental results minimal, consistency measurements are performed on only 1% of all objects.

2.3.2 Ordering Guarantees

Knowledge about the order in which writes become visible can be derived from knowledge about stale reads. In [BVF+12], Bailis et al. capture the probability of *MRC* in the PBS prediction model. Both Wada et al. [WFZ+11] and Bermbach et al. [BT14] compute MRC violations on the basis of the log files generated during their staleness measurements. In addition, Bermbach et al. also examine their experiment logs for violations of *MWC* and *RYWC*. To our knowledge, no method for the quantification of *WFRC* violations has been proposed so far.

2.3.3 Transactions

Recently, a wealth of scalable transactional datastores have been proposed and implemented. Examples of these systems are large-scale distributed systems like Megastore, Spanner, Percolator and F1 from Google, key-value stores enhanced by atomic multi-key transactions like COPS, Granole, G-Store and Hyperdex, as well as more general-purpose commit protocols and transaction managers like MDCC or Omid.

Bailis et al. examine the design space for systems that provide ACID guarantees and high availability at the same time in [BDF+13]. They identify Read Committed as an isolation level that is often used in traditional single-node database systems and can be achieved in a highly available transactional datastore. Furthermore, the authors state that lost updates and write skew as well as concurrent updates and potentially unbounded staleness can never be prevented in a highly available system.

To fill the gap between classic SQL-based transactional benchmarks (e.g. TPC-C) and simple cloud service benchmarks, **YCSB+T** [DFN$^+$14] was proposed. It adds to the four tiers defined in the original YCSB contribution (performance, scalability, availability and replication) by introducing two new tiers *transactional overhead* and *consistency*. The *Transactional overhead* tier measures the latency of transactional operations (read, scan, insert, update, delete, read-and-modify) and transaction demarcation (start, abort, commit). To achieve this, a so-called *Closed Economy Workload (CEW)* is defined. It simulates bank account transactions in a closed system where money neither enters nor exits. This workload executes operations similar to YCSB, but wrapped in a single transactional context. For instance, `doTransactionalReadModifyWrite` reads two account records, transfers some money from one to the other and writes both records back. To achieve this, the central contract between YCSB and the database, the *DB interface*, is enhanced by (optional) transactional methods. The amount of concurrent transactions is determined by YCSB's *threads* parameter which defines how many threads execute the workload in parallel. To ensure that no anomalies (e.g. lost updates) occur during workload execution, a *validation phase* is executed after the transaction (consistency tier). The application-defined validation method takes the database state as an input and calculates an *anomaly score*. For the CEW, this score is simply defined as the difference between the initial and final sum of all accounts normalized by the amount of executed operations. This score has a major problem: Not only does it not detect dirty reads and non-repeatable reads, it also catches only a fraction of all lost updates, as lost updates can easily occur without any changes to the sum of all account balances.

The evaluation of YCSB+T demonstrates the usage for one particular system, but lacks a comparison of different transactional datastores. Benchmarking different scale-out transactional systems remains an important open issue in this field. YCSB+T furthermore does not detect transaction anomalies, it is limited to verifying state-based consistency constraints. While measuring anomalies (i.e. isolation level compliance) has been studied for single-server scenarios [FGA09], deriving a scheme for distributed transactional datastores thus remains an open issue. Another open challenge is the inclusion of the *availability* and *replication* tiers for transactional benchmarking. To decouple concurrency from transaction sizes and amounts, it would also be desirable to construct a benchmark which allows for more complex configurations than a one-to-one mapping between threads and transactions.

2.3.4 Consistency in the Face of Partitions

Informally, partition tolerance is the ability of a system to sustain operation in the presence of message loss between the nodes. As partitions are generally unavoidable, partition tolerance is guaranteed by virtually all NoSQL databases.

While availability in failure scenarios has already been addressed by several studies, only little work has been done on whether NoSQL databases keep their promise to remain consistent during and after partitions. In his *Call Me Maybe* blog post series[6], Kyle Kingsbury examines several widely used distributed systems and their behaviour during network partitions, revealing that many do not comply with their marketing claims and actually lose data where they should not. To this end, he presents *Jepsen*[7], an open-source project that facilitates injecting failures and running tests in a virtualised distributed environment. Through his work, Kingsbury uncovers misunderstandings and implementation errors in existing database systems.

3 Our Approach

To provide strict bounds on staleness, we are working on an approach that takes up the idea of YCSB++ to provide a lower bound for the staleness window and extends it by measuring an upper bound. The basic concept of our approach is illustrated in Figure 3:

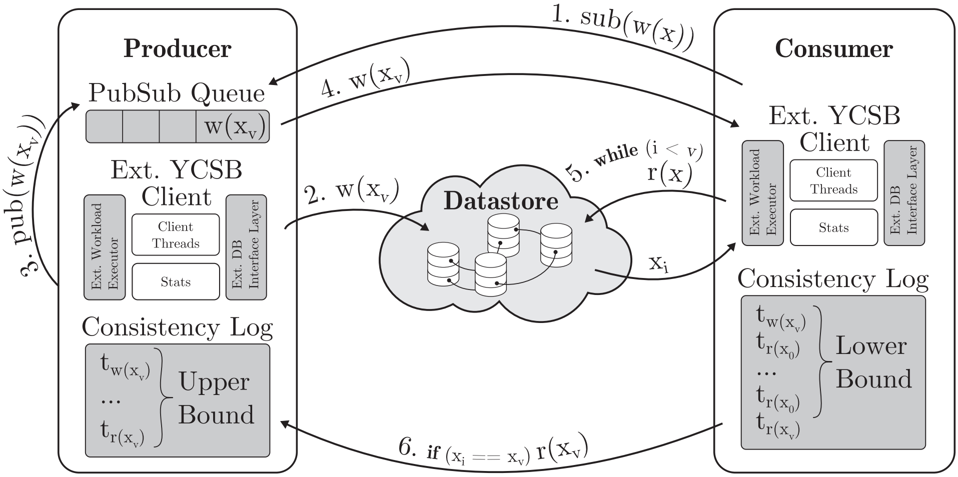

Figure 3: Our multi-client consistency measuring approach employs decentralized PubSub queuing to measure lower and upper bounds for staleness.

Similar to the YCSB++ approach, our experimental setup comprises at least one producer and one consumer client as well as, of course, one datastore. One physical machine can take the role of the producer for one data item and the role of the consumer for another. Furthermore, several consumer clients can be used to measure the staleness window of the same data item. Instead of handling communication through centralised messaging (ZooKeeper in the YCSB++ approach), we use decentralized client-side messaging. Prior to the start of an experiment, a consumer subscribes (1.) to be notified as soon as a specific data item is updated by the producer. The producer writes (2.) the data item and publishes (3.) the write to its local queue, so that every consumer is notified (4.) to repeatedly read (5.) the item, until the new value is observed. If a consumer reads the new version of a

[6] *Call Me Maybe* blog post series: `http://aphyr.com/tags/jepsen`.

[7] Jepsen: `https://github.com/aphyr/jepsen`

stale value, he notifies the producer (6.). This allows us to compute an upper bound for the staleness window of this particular read operation as the difference between the producer's timestamps directly after the consumer notification (6.) and directly before the write (2.).
Apart from preventing contention, reducing latency and simplifying the process of setting up an experiment in comparison to YCSB++, using local messaging queues eliminates network latency altogether, if producer and consumer share the same physical machine. Furthermore, our approach does not rely on clock synchronisation and provides bounds for the actual inconsistency that a client experienced. It should be noted, though, that a valid upper bound can only be yielded under the assumption of MRC; in the absence of MRC guarantees, the consumer client can request (.5) the data item for a longer time in order to increase the probability of observing MRC violations, if they exist.

4 Conclusion and Open Challenges

NoSQL OLTP benchmarking is an active research topic and the boundaries of what is achievable in the field of distributed databases are being probed by both scientists and practitioners. Arguably, the most popular and most widely accepted OLTP benchmark for NoSQL databases is YCSB which facilitates measuring operational throughput and request latency for generic CRUD workload mixes. YCSB's obvious strong points are an easy-to-implement database interface, easy-to-use design and easy-to-extend architecture. On the other hand, its highly generic design can also be seen as a drawback since it does not account for functionality beyond simple CRUD and therefore does not capture the performance of datastores well that offer more sophisticated operations. It is also possible to model real-world applications like BG does, but this restricts applicability.

Several aspects of availability such as the effect of replication on steady-state performance or system performance during node failure and recovery have already been addressed in experimental evaluation. The differences between the available replication strategies and their respective configurations, however, have not been studied to full extent as far as we are aware.

As data-centric consistency cannot be measured without internal knowledge of the storage system, only client-centric approaches seem viable for generic benchmarking frameworks. Client-centric consistency in distributed systems can be measured along two dimensions: Staleness describes the time during which an acknowledged write is not applied and ordering refers to the order in which writes become visible to clients.
An inherent issue with benchmarking consistency in a distributed system, however, is that distributed measurement is also subject to imprecision through network latency and clock synchronisation. Several approaches to deal with these issues can be found in the literature: Having reader and writer of the data item under consideration share the same machine eliminates the communication delay, but also may lead to unrealistic results. Distributing reader and writer over separate physical machines, in contrast, necessitates clock synchronisation or leads to latency values that either include or ignore network delays. One rationale in workload design is to measure only observed inconsistency, while the other is to introduce artificial operations to increase the precision of the results. An interesting

question is whether the latter approach can actually lead to an observer effect.
To our knowledge, only little work has been done on the quantification of ordering guarantee violations in distributed databases. Until recently, the transactional consistency provided by a datastore has not been addressed by experimental evaluation. YCSB+T is an important first step towards transactional benchmarking of scale-out datastores and provides a useful basis for quantifying the overhead introduced by wrapping CRUD operations in a transaction. However, the inclusion of the availability and replication tiers for transactional benchmarking which have already been proposed in the original YCSB paper remain open issues.

Ongoing work shows that aspects of performance like availability and consistency can be quantified, but individual experiments only cover small parts of the vast space of possible experiment configurations. For example, the impact of different failure scenarios or (wide-area) replication strategies on both availability and consistency still need to be examined. Further, the integration of existing approaches into a comprehensive and widely applicable benchmarking suite is an important goal of future work.

References

[Aba12] D. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, Feb 2012.

[BDF$^+$13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, et al. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[BG13] Sumita Barahmand and Shahram Ghandeharizadeh. BG A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.

[BGH$^+$13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, et al. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[Bre00] Eric A. Brewer. Towards Robust Distributed Systems., 2000.

[BT11] David Bermbach and Stefan 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*, MW4SOC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[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.

[BVF$^+$12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, et al. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[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.

[CST^{+}10] Brian F. Cooper, Adam Silberstein, Erwin Tam, et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[DFN^{+}14] Akon Dey, Alan Fekete, Raghunath Nambiar, et al. YCSB+T: Benchmarking Web-scale Transactional Databases. In *Proceedings of International Workshop on Cloud Data Management (CloudDB'14)*, Chicago, USA, 2014.

[Eng13] Ben Engber. How to Compare NoSQL Databases: Determining True Performance and Recoverability Metrics For Real-World Use Cases. Presentation at NoSQL matters 2013, 2013.

[FGA09] Alan Fekete, Shirley N. Goldrei, and Jorge Pérez Asenjo. Quantifying Isolation Anomalies. *Proc. VLDB Endow.*, 2(1):467–478, August 2009.

[FMdA^{+}13] Alessandro Gustavo Fior, Jorge Augusto Meira, Eduardo Cunha de Almeida, et al. Under Pressure Benchmark for DDBMS Availability. *JIDM*, 4(3):266–278, 2013.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[GLS11] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing Consistency Properties for Fun and Profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 197–206, New York, NY, USA, 2011. ACM.

[HKJ^{+}10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, et al. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[LFK^{+}11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, et al. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[NE13] Denis Nelubin and Ben Engber. NoSQL Failover Characteristics: Aerospike, Cassandra, Couchbase, MongoDB. Technical report, Thumbtack Technology, 25 Broadway, Floor 9, New York, 2013.

[PPR^{+}11] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 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, New York, NY, USA, 2011. ACM.

[RGA^{+}12] Muntasir Raihan Rahman, Wojciech Golab, Alvin AuYoung, et al. Toward a Principled Framework for Benchmarking Consistency. In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[WFZ^{+}11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR'11*, pages 134–143, 2011.