B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 215

Coordinated Omission in NoSQL Database Benchmarking

Steffen Friedrich[1] Wolfram Wingerath[2] Norbert Ritter[3]

Abstract: Database system benchmark frameworks like the Yahoo! Cloud Serving Benchmark (YCSB) play a crucial role in the process of developing and tuning data management systems. YCSB has become the de facto standard for performance evaluation of scalable NoSQL database systems. However, its initial design is prone to skipping important latency measurements. This phenomenon is known as the coordinated omission problem and occurs in almost all load generators and monitoring tools. A recent revision of the YCSB code base addresses this particular problem, but does not actually solve it. In this paper we present the latency measurement scheme of NoSQLMark, our own YCSB-based scalable benchmark framework that completely avoids coordinated omission and show that NoSQLMark produces more accurate results using our validation tool SickStore and the distributed data store Cassandra.

Keywords: Database Benchmarks, Coordinated Omission, NoSQL

1 Introduction

In recent years, a lot of new distributed database management technologies broadly classified under the term NoSQL databases have emerged for large data intensive applications. Since the NoSQL landscape is very diverse [Ge16], decision makers have to rely on performance benchmarks in addition to functional requirements to select the appropriate data management solution for their application needs. Traditional relational database systems are evaluated with industry standard benchmarks like TPC-C [Tr05] and TPC-E [Tr07]. These are distinguished by the fact that they model particular applications and run workloads with queries and updates in the context of transactions, whereas the data integrity is supposed to be verified during the benchmark. Because NoSQL databases sacrifice consistency guarantees, transactional support and query capabilities, a new benchmark, the Yahoo! Cloud Serving Benchmark (YCSB) [Co10], has become the de facto standard, which does not model a real-world application, but examines a wide range of workload characteristics with customisable mixes of read/write operations. To support any NoSQL database, YCSB is limited to a CRUD interface only.

Nowadays, almost every new research on the NoSQL domain has been experimentally evaluated with the help of YCSB and there have been many extensions developed for it [Fr14]. Often overlooked was the underlying system model of its load generator, until recently when the *coordinated omission problem* attracted great interest among practitioners. The problem was named by Gile Tene, CTO and co-founder at Azul Systems, who describes

[1] University of Hamburg, ISYS, Vogt-Kölln-Straße 30, Hamburg, Germany, friedrich@informatik.uni-hamburg.de
[2] University of Hamburg, ISYS, Vogt-Kölln-Straße 30, Hamburg, Germany, wingerath@informatik.uni-hamburg.de
[3] University of Hamburg, ISYS, Vogt-Kölln-Straße 30, Hamburg, Germany, ritter@informatik.uni-hamburg.de

it in his talks on "How not to measure latency" [Te16]. We explain the coordinated omission problem in Section 2. Its great attention led to an extension of YCSB that tries to counteract the problem. Therefore, in addition to the normal latency values, corrected, *intended* latency values are computed. We describe this correction in Section 2.1 and explain how we can avoid the problem at all with our scalable benchmark framework NoSQLMark in Section 3. To investigate the effect of the coordinated omission problem, we extend our database system SickStore in Section 4 and evaluate how the intended latency values differ from those with coordinated omission avoidance in Section 5. We finish the paper with a summary and concluding remarks in Section 6.

2 The Coordinated Omission Problem

The Java code snippet in Listing 1 shows how YCSB generates load and measures response times. To issue requests with a given target throughput, the required wait time between two operations is computed (1). After measuring the latency for one request (5-8), the measurement thread sleeps until the computed deadline is reached (12-16). Since the call to the database system under test is done synchronously, this methodology only works well as long as the response time falls within the desired time window. For illustration, consider the example measurements depicted in Figure 1.

```
1  _targetOpsTickNanos = (long) (1_000_000_000 / target)
2  long overallStartTime = System.nanoTime();
3  while (_opsdone < _opcount) {
4    long startTime = System.nanoTime();
5    Status status = _db.read(table,key,fields,result);
6    long endTime = System.nanoTime();
7    _measurements.measure("READ",
         (int)((endTime-startTime)/1000));
8
9    _opsdone++;
10
11   long deadline = overallStartTime + _opsdone *
         _targetOpsTickNanos;
12   long now = System.nanoTime();
13   while((now = System.nanoTime()) < deadline) {
14     LockSupport.parkNanos(deadline - now);
15   }
16 }
```

List. 1: A code snippet, similar to YCSBs source code, that shows how load generators typically measure latency.

To illustrate the problem, we run a workload with one client thread, a target throughput of 10 ops/sec and configure our database SickStore (introduced in Section 4) to simulate a *hiccup* (disruptions and delays caused by garbage collection, context switches, cache

buffer flushes to disk, etc.) of one second after 10 seconds run time. According to Listing 1, we want the client thread to issue one request each 100 milliseconds. Request 1 to 9 take less than 50µs each, request 10 takes one second. The subsequent request again takes only around 50µs because it does not happen at its designated time. In this manner, the database system under test delays requests that would have been made during the synchronous call, which leads to the coordinated omission of relevant measurements such that the overall measurement results do not reflect database hiccups very well. This obscuration becomes worse if the results are reported as the mean latency only.

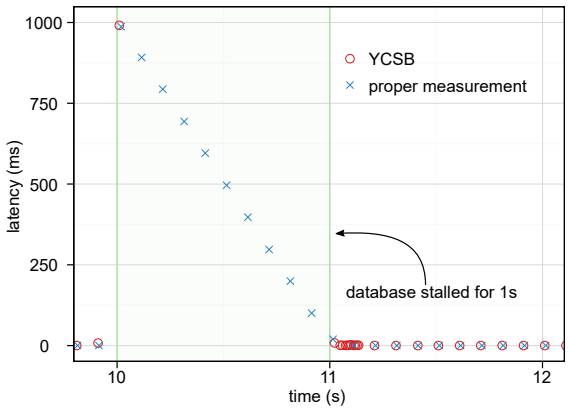
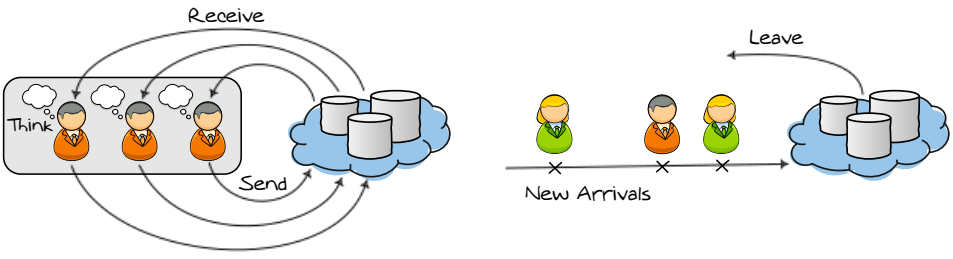


Fig. 1: Measurements that illustrate the coordinated omission problem.

According to Schröder et. al. [SWHB06], YCSB follows a *closed system model*: There is some fixed number of users that repeatedly request the system, receive the response and then "think" for some amount of time. A new request is only sent after the completion of the previous one. However, YCSB focuses on workloads web applications place on cloud data systems. Therefore, we actually want to measure the response time for user requests that appear independently such that requests arrive in an unbounded fashion even if the database system stalled for a longer period of time. This corresponds to an *open system model*, where users arrive with an average arrival rate λ and a new request is only triggered by a new user arrival, independently of request completions.



(a) Closed system model (b) Open system model

Fig. 2: Illustrations of the closed and open system models.

We assume that many developers do not consciously take this design difference into account. Schröder et. al. came to a similar conclusion and found that most of the benchmark systems that they investigated, including the relational database system benchmarks TPC-C and TPC-W, follow a closed system model. But developers in general often do not mention anything about the underlying system model in their documentations. An exception is Sumita Barahmand who states in her PhD thesis on benchmarking interactive social networking actions [Ba14] that her framework BG implements a closed system model, even though she, too, considers the open model more realistic for a social networking site (or websites in general). But she leaves this implementation open for future research.

2.1 Coordinated Omission Correction in YCSB

With the release 0.2.0 RC1 of YCSB in June 2015, some changes regarding the coordinated omission problem were introduced by Nitsan Wakart[2]. First of all, the histogram library called HdrHistogram[3] was added. In contrast to the standard histogram implementation with a fixed number of buckets, the HdrHistogram can measure a dynamic range of measurement values with configurable value precision. Additionally, the library provides a loss-free compressed serialization which allows the reconstruction of the complete histogram, for example to export data for plotting percentile curves. However, to overcome the coordinated omission problem the most important change is the implementation of the intended measurement interval. The idea is to correct the latency values by setting the measurement start time to the designated time of the request. Consider again Listing 1, the intended latency of the actual request can be measured by using the deadline of the preceding request as start time:

```
measure("Intended-READ", (int)((endTime - _precedingDeadline)/1000));
```

This may appear to solve the problem, but the database is still able to influence the actual time of the request which may affect the measurement results. Because all pending requests (i.e. their calculated deadlines lie in the past) are sent immediately after a delayed response, a very high load is generated temporarily. The big difference to an open system model is that the requests are queued on the client side, whereas in the open model a request could be sent to and processed by the database, even if the database is still processing the previous one (e.g. due to multithreading or load balancing across multiple nodes). Intuitively, we would therefore suspect that the corrected latency values tend to be higher than with an open system model. We examine this hypothesis experimentally in Section 5.

3 Coordinated Omission Avoidance in NoSQLMark

During our research on staleness-based consistency measurement methods [Wi15], we decided to develop our own benchmarking framework NoSQLMark on top of YCSB, to

[2] Changes in YCSB regarding the coordinated omission problem: https://github.com/brianfrankcooper/YCSB/blob/0.2.0-RC1/core/CHANGES.md
[3] HdrHistogram library for Java: https://github.com/HdrHistogram/HdrHistogram

overcome several of it's shortcomings. NoSQLMark is written in Scala with the Akka toolkit and thus allows to easely scale a workload on different benchmark nodes. You no longer have to start YCSB manually on different servers nor do you have to aggregate the different measurement results by your own. Since Java libraries can be used directly in Scala code we can depend on YCSB to support all of its database bindings as well. Furthermore, the actor model allows us to implement more complex workloads that require communication between actors like for example in our proposed consistency measurement method [Fr14]. To avoid coordinated omission, we implement an open system model by sending our requests in an asynchronous fashion. The Scala code snippet in Listing 2 shows how an actor generates load and measures latency.

```
1  implicit val executionContext = context.system.
2          dispatchers.lookup("blocking-io-dispatcher")
3
4  case DoOperation => {
5    val operation: Operation = workload.nextOperation
6    val startTime = System.nanoTime
7    val future = Future {
8      sendRequest(operation)
9    }
10   future.onComplete {
11     case Success(status) => {
12       val endTime = System.nanoTime
13       measurementActor ! Measure(operation.name,
             status, (endTime - startTime) / 1000)
14     }
15     case Failure(ex) => {
16       log.error(ex, "Error occured during operation
             {}", operation.name)
17     }
18   }
19   _opsdone += 1
20   if (_opsdone < _opcount) scheduleNextOperation()
21   else supervisor ! WorkIsDone(workerID, jobID)
22 }
```

List. 2: Scala code snipped for asynchronous load generation to implement an open system model in NoSQLMark.

Since an actor processes information by means of handling asynchronous messages, each Akka actor has to implement the receive method which should define a series of case statements that defines which messages an actor can handle. Thus the code snippet shows how our actor (called *worker*) handles the message of type `DoOperation`. The critical difference to Listing 1 is that we use a scala `Future` to send the actual request asynchronously (7-9). Note that we create a dedicated execution context for blocking i/o and have tried various

executor implementations / configurations[4] (1-2). Next, we register a callback on the future by using the `onComplete` method, which is applied to the value of type `Success[Status]` if the future completes successfully, or to a value of type `Failure[Throwable]` otherwise (10-18). If successful, we send the computed latency to the measurement actor (11-14). The `scheduleNextOperation` method schedules a new message of type `DoOperation` to be send to the actor himself (20). Thus, the actor remains more or less reactive for another message, e.g. an abort message by the user. At the end, a `WorkIsDone` message is sent to the supervisor actor (21), who still have to wait for the outstanding measurements by the measurement actor. The entire source code will be released in open source and we will publish further details on NoSQLMark's architecture and future development at the official project page[5].

4 Designing SickStore for Coordinated Omission

We introduced SickStore, the **s**ingle-node **inc**onsistent **k**ey-value **store** to validate existing staleness benchmarks [Wi15]. For this purpose, SickStore's design enables it to simulate a replica set consisting of multiple virtual storage nodes that produce stale values, but at the same time, it provides consistent knowledge about the system state at any point in time. This is simplified by running the server in a single thread. Nevertheless, in order to allow the simulation of latencies, the server returns the calculated latency values to the client library, which blocks the calling thread for the corresponding remaining time.
In order to investigate the problem of coordinated omission, we have extended SickStore to simulate a maximum throughput λ_{max} and database hiccups, both configurable per virtual node. Again, the server process should not really be slowed down or stopped. Therefore, we simulate a request queue by introducing a counter C for the number of outstanding requests in the system. Let

$$\begin{aligned} t_i &= \text{ timestamp at which request } i \text{ was received by the server;} \\ \Delta_{i,j} &= t_j - t_i \text{ the time period between request } i \text{ and } j \text{ where } t_i < t_j; \\ C(t_i) &= \text{ the number of outstanding requests at time } t_i;. \\ T_i &= \text{ the waiting time of request } i \text{ in the system} \end{aligned}$$

In order to simulate the maximum throughput, for each request i we compute

$$C(t_i) = max(0\ ,\ C(t_{i-1}) - \lambda_{max} \times \Delta_{i-1,i})\ , \tag{1}$$

where $i-1$ is the direct predecessor to i. Note if i is the first request then C must be equal to 0. Then we compute $T_i = C(t_i)\ /\ \lambda_{max}$ and finally add one to the counter C before returning T_i to the client library. To also simulate a hiccup, we only have to increase the counter C by

[4] We ended up with a similar configuration as suggested in the Akka docs, but with a high pool-size-factor instead of a fixed-pool-size, to get a cached thread pool with a high limit and an unbounded queue: http://doc.akka.io/docs/akka/current/scala/dispatchers.html#More_dispatcher_configuration_examples

[5] NoSQLMark can be obtained from http://nosqlmark.informatik.uni-hamburg.de/

the appropriate number of requests. Let

$$t_h = \text{ the desired start time of the hiccup;}$$
$$H = \text{ the desired hiccup duration.}$$

Then equation (1) becomes

$$C(t_i) = \begin{cases} \max(0\ ,\ C(t_{i-1}) - \lambda_{max} \times \Delta_{i-1,i} + \lambda_{max} \times H), & \text{if } t_i < t_h <= t_j \\ \max(0\ ,\ C(t_{i-1}) - \lambda_{max} \times \Delta_{i-1,i}), & \text{otherwise} \end{cases} \quad (2)$$

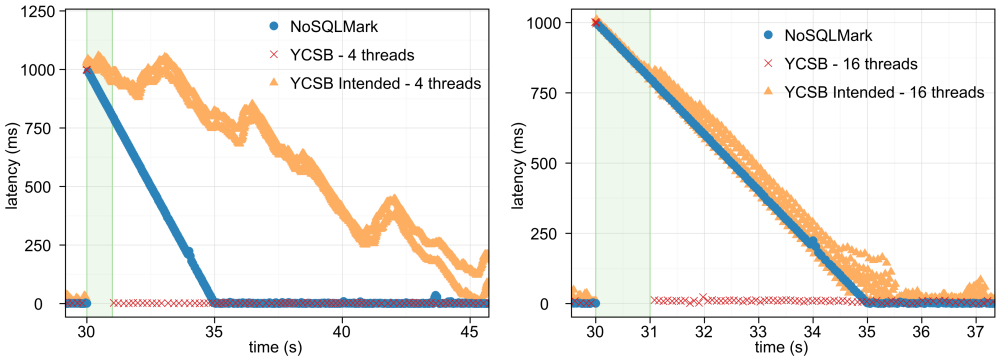
Finally, each calling client thread has to pause for the duration $l + T_i$, where l is the calculated network latency by SickStore's latency generator.

5 Experimental Evaluation

We proceed to experimentally demonstrate that the latency values corrected by the intended measurement interval in YCSB greatly differ from those measured by our implementation of an open system model. This is confirmed by the exemplifying simulation of a hiccup with SickStore, as well as in our experiments with Cassandra.

5.1 Coordinated Omission Simulation with SickStore

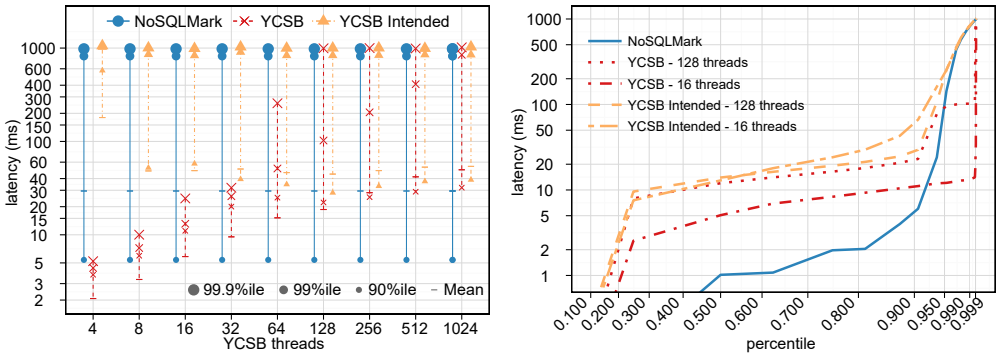
To get an insight into how the measurement results behave in a hiccup situation, we have conducted a series of micro-benchmarks with SickStore. For the simulation, we used one machine with 2.60 Ghz Quad-Core Intel i7-6700HQ processor and 16 GB memory.We run a workload with a target throughput of 1000 ops/sec, 90000 operations and configure SickStore to simulate a hiccup of one second after 30 seconds run time. Thus, each micro-benchmark runs 90 seconds, which is sufficient for each run to stabilize the latency values to a level as before the hiccup. In the first series of experiments we increase the number of YCSB client threads and set the maximum throughput at SickStore to 1250 ops/sec. The two time series plots in Figure 3 illustrate how the number of YCSB client threads affects the measurement behavior. We see that with NoSQLMark, the latency values are exactly stabilized at the point as we would expect with the configured maximum throughput and hiccup duration. For YCSB, the number of requests during the hiccup is limited by the number of threads and many latency values are omitted, which are finally to be corrected by the intended measurement interval. Surprisingly, the intended measurements stabilize very late for 4 threads, which is already much better with 16 threads. In Figure 4(a), we see some percentiles and mean values for the latencies with different YCSB thread numbers. Even if the number of threads increases further, the intended values are higher than those of NoSQLMark. This is particularly true for the mean value and the 90-percentiles, which are approaching up to 128 threads ever closer to the results of NoSQLMark but increase again as of 256 threads. It is noticeable that the uncorrected mean value approaches the mean value of NoSQLMark with increasing thread count as well. In addition, as the number of threads increases, the corrected and uncorrected latency values of YCSB approach each



(a) 4 YCSB threads (b) 16 YCSB threads

Fig. 3: Time series plots of two runs with different numbers of YCSB threads.

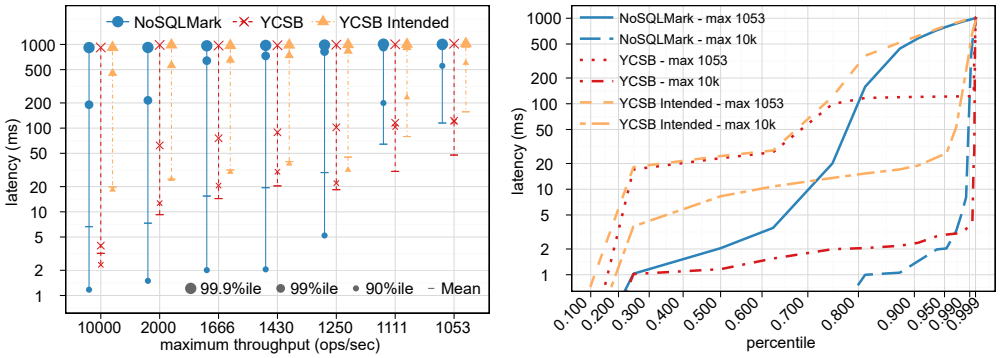
other, which becomes even more apparent when viewing the entire percentile spectrum in Figure 4(b). Incidentally, it also shows that the latencies of the lower percentiles are one order of magnitude higher than those of NoSQLMark. Next, we set the number of threads



(a) Overview (b) Percentile spectrum for 16 and 128 YCSB threads.

Fig. 4: Simulation with different numbers of YCSB threads.

to 128 and vary the maximum throughput. Figure 5 shows again the overview and the entire percentile spectrum for two selected experiments. For all three measurement approaches, it can, of course, be observed that the latency values increase with higher system load. But in principle, the measurement results differ considerably, even when the system is under different load conditions. In Figure 5(b) one can see how all percentile curves shift to the upper left corner of the spectrum, with higher system utilization. Although the difference in the 90th and higher percentiles appears to be smaller, it is significant for smaller percentiles (e.g. 200ms in the 80th percentile between NoSQLMark and YCSB Intended). In summary, our simulation of a single hiccup shows that the fundamentally different system models of load generation also result in a completely different percentile spectrum of the latency values.



(a) Overview

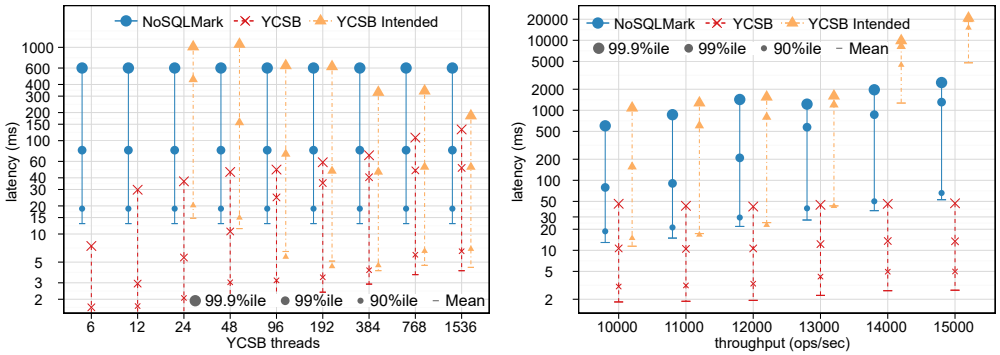
(b) Percentile spectrum for max 1053 and 10k ops/sec.

Fig. 5: Changing SickStore's maximum throughput.

5.2 Elasticity Benchmark with Cassandra

Kuhlenkamp et. al. have recently repeated scalability and elasticity experiments of previous research with YCSB [KKR14]. Since their results do not include the intended latency values of YCSB, it is a good idea to repeat an experiment to see the difference between the two system models of the load generators and the intended latencies. We performed the benchmark with a simpler configuration by running the workload on a single Cassandra node and adding another node to the cluster after 5 minutes. For each node, we used instances of our private OpenStack cloud with 4 vCPUs and 16 GB of memory and one instance with 6 vCPUs and 63 GB memory for the benchmark. Each instance runs on a separate server (2,1 GHz Hexa-Core Intel Xeon E5-2620v2, 15MB Cache, 64 GB memory, 6 disk RAID-0 array and gigabit ethernet). The Cassandra node is initially loaded with 10 million rows, i.e., the node manages 10 GB. We perform the experiments with YCSB workload A with a Zipfian request distribution and add the second node to the cluster after 5 minutes of runtime without a data streaming throughput limit. At a target throughput of 10 000 ops/sec, the second node starts serving after roughly 5 minutes, i.e. after 10 minutes run time the latency stabilizes, which is why we run each experiment for a maximum duration of 15 minutes.

Figure 6 shows the results at a glance, (a) for an increasing number of threads (adapted to 6 cores) with a target throughput of 10k ops/sec, and (b) for 48-thread increasing target throughput. 15k ops/sec was the maximum throughput that could be achieved. We can again observe that the measurement results of YCSB and YCSB Intended are becoming more and more similar with increasing number of threads. The intended latency values for the represented percentiles even fall below the values of NoSQLMark, starting at 96 threads. However, 10k ops/sec is also only 66.6% of the maximum achievable load. On the other hand, we see that the difference between the three approaches increases drastically as the load increases. Nevertheless, the same observations as in our simulation with SickStore are also reflected in these results.



(a) Different numbers of YCSB threads (10k ops/sec) (b) Different target throughputs (48 YCSB threads)

Fig. 6: Percentiles and mean value for the elasticity benchmark with Cassandra.

6 Concluding Remarks

In this paper, we described the coordinated omission problem and how it relates to the system model of load generators. Just about all benchmark frameworks are based on a closed system model and thus have the coordinated omission problem. We described how YCSB tries to correct the latency values by introducing the intended measurement interval. Afterwards we showed how an open system model was implemented in NoSQLMark and how we extended the simulation tool SickStore to investigate the coordinated omission problem. Finally, we conducted several experiments which showed that the different system models lead to completely different measurement results, even the intended latency values greatly differ from those of the open system model.
From our results, we conclude that developers of a benchmark framework should be more concerned about the underlying system model, because the type of load generation can not be corrected afterwards. For this reason, we plan to refine the load generator of NoSQLMark in the future, by generating the load using different distributions such as Possion.

References

[Ba14] Barahmand, Sumita: Benchmarking Interactive Social Networking Actions. PhD thesis, USC Computer Science Department, Los Angeles, California 90089-0781, 2014.

[Co10] Cooper, Brian F.; Silberstein, Adam; Tam, Erwin; Ramakrishnan, Raghu; Sears, Russell: Benchmarking Cloud Serving Systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC '10, ACM, New York, NY, USA, pp. 143–154, 2010.

[Fr14] Friedrich, Steffen; Wingerath, Wolfram; Gessert, Felix; Ritter, Norbert: NoSQL OLTP Benchmarking: A Survey. In: 44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland. pp. 693–704, 2014.

[Ge16] Gessert, Felix; Wingerath, Wolfram; Friedrich, Steffen; Ritter, Norbert: NoSQL database systems: a survey and decision guidance. Computer Science - Research and Development, pp. 1–13, 2016.

[KKR14] Kuhlenkamp, Jörn; Klems, Markus; Röss, Oliver: Benchmarking Scalability and Elasticity of Distributed Database Systems. Proc. VLDB Endow., 7(12):1219–1230, August 2014.

[SWHB06] Schroeder, Bianca; Wierman, Adam; Harchol-Balter, Mor: Open Versus Closed: A Cautionary Tale. In: Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3. NSDI'06, USENIX Association, Berkeley, CA, USA, pp. 18–18, 2006.

[Te16] Tene, Gil: , How NOT to Measure Latency. https://www.infoq.com/presentations/latency-response-time, March 2016.

[Tr05] Transaction Processing Performance Council: , TPC-C Benchmark Specification. http://www.tpc.org/tpcc, 2005.

[Tr07] Transaction Processing Performance Council: , TPC-E Benchmark Specification. http://www.tpc.org/tpce, 2007.

[Wi15] Wingerath, Wolfram; Friedrich, Steffen; Gessert, Felix; Ritter, Norbert: Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking. In: Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings. pp. 351–360, 2015.