Tell-Tale Tail Latencies: Pitfalls and Perils in Database Benchmarking

Michael Fruth[1][0000−0003−2933−5093], Stefanie Scherzinger[1], Wolfgang Mauerer[2,3][0000−0002−9765−8313], and Ralf Ramsauer[2]

[1] University of Passau, 94032 Passau, Germany
{michael.fruth,stefanie.scherzinger}@uni-passau.de
[2] Technical University of Applied Sciences Regensburg, 93058 Regensburg, Germany
{wolfgang.mauerer,ralf.ramsauer}@othr.de
[3] Siemens AG, Corporate Research, Otto-Hahn-Ring 6, 81739 Munich, Germany

Abstract. The performance of database systems is usually characterised by their average-case (*i.e.*, throughput) behaviour in standardised or de-facto standard benchmarks like TPC-X or YCSB. While tails of the latency (*i.e.*, response time) distribution receive considerably less attention, they have been identified as threat to overall system performance: In large-scale systems, even a fraction of requests delayed can build up into delays perceivable by end users.
To eliminate large tail latencies from database systems, the ability to faithfully record them, and likewise pinpoint them to the root causes, is imminently required. In this paper, we address the challenge of measuring tail latencies using standard benchmarks, and identify subtle perils and pitfalls. In particular, we demonstrate how standard benchmarking approaches can substantially distort tail latency observations, and discuss how the discovery of such problems is inhibited by the common focus on throughput performance. We make a case for purposefully re-designing database benchmarking harnesses based on these observations to arrive at faithful characterisations of databases from multiple important angles.

Keywords: Database benchmarks · Tail latencies · Benchmark harness.

1 Introduction

Measuring performance is an essential ingredient of evaluating and optimising database management systems, and a large fraction of published research (*e.g.*, [3, 19, 20, 25, 26, 29, 33]) is driven by guidance from the collection of benchmarks provided by the Transaction Processing Performance Council [48], or commercial de-facto standards like the Yahoo! Cloud Serving Benchmark (YCSB) [10].

These benchmarks usually focus on measuring *throughput* (*i.e.*, number of operations performed in a given time interval), or latency (*i.e.*, time from submitting a request to receiving the result, usually characterised by the 95th or 99th percentile [**?**] of the response time distribution). However, it is known that high latency episodes rarer than events in the 99th percentile may severely impact the

whole-system performance [12], including important use-cases like interactive web search [4]—even if they do not receive much attention in standard performance evaluations. In this article, we focus on properly characterising tail latencies in database benchmarking, and unearth shortcomings in popular benchmark setups.

We find that tail latencies observed in the ubiquitous TPC-C or YCSB benchmarks for commonly used databases often fall into the millisecond range, but are caused by the benchmarking process itself. Since systemic optimisation efforts are expected to require targeting microsecond latencies, aptly termed "killer microseconds" by Barroso *et al.* [2], is seems evident that non-productive perturbations that exceed such delays by three orders of magnitude make it impossible to obtain a faithful characterisation of database performance.

We show that a popular, Java-based benchmark harness, that is, the software setting up and executing database benchmarks [34], records latencies that were actually imposed by its own execution environment, in particular garbage collection in the Java virtual machine (JVM). That is, we show that significant noise is caused by the benchmark harness itself, disturbing the measurements. However, latencies must but pinpointed to their actual source before targeted improvements can unfold their impact. If measured latencies are not identified as being caused by the measurement setup, developers will inevitably fail to pin them down, and therefore cannot address them properly.

Contributions. In this article, we claim the following contributions, based on measuring latencies in database query evaluation, using the commonly used benchmark harness *oltpbench* [11, 13] with two well-accepted benchmarks (YCSB and TPC-C) on mature database management systems (MySQL and PostgreSQL), capturing throughput and tail latencies:

- We show that seemingly irrelevant technical details like the choice of java virtual machine (and even the particular garbage collection mechanism) of the benchmark dispatcher can severely distort tail latencies, increasing maximum latencies by up to several orders of magnitude, while the usually considered quantities median, 95th and 99th percent latencies remain largely unperturbed.
- We carefully separate systemic noise (such as caused by the system software stack) from the noise intrinsic to the benchmarking harness and the database system. We succeed in identifying and isolating the latencies introduced by the the garbage collector managing the memory for the benchmark harness.
- Based on specially crafted dummy components, we carefully characterise upper *and* lower bounds on the influence of the measurement infrastructure on the measurement itself, enabling researchers and developers to distinguish between relevant results and source of non-productive perturbations caused by the measurement itself.

Overall, we systematically build a case for adapting benchmarking harnesses to faithfully capturing tail latencies.

Structure. This paper is organised as follows. We review the preliminaries in Section 2. We then present our experiments in Section 3, which we further discuss

in Section 4. We state possible threats to validity in Section 5, and review related work in Section 6. We conclude with Section 7.

2 Preliminaries

2.1 Database Benchmarks

TPC-C [47], defined in 1992, is an industry-standard measure for OLTP workloads. The benchmark models order processing in warehouses.

The *Yahoo! Cloud Serving Benchmark* (YCSB) [10] is an established big data benchmark. YCSB handles lightweight transactions, as most operations access single record only. This results in low latencies of requests compared to a benchmark such as TPC-C.

The No Operation (NoOp) benchmark is a simplistic baseline for comparison, as it sends an empty statement (*e.g.*, the semicolon command for PostgreSQL), which has to be acknowledged by the database system. NoOp benchmarks can be used to quantify the raw measurement overhead of the benchmark harness. The NoOp benchmark also can be interpreted to represent the minimum client-server *round-trip time* of a statement.

2.2 The OLTPBench Benchmark Harness

A benchmark harness is a toolsuite that provides the functionality to benchmark a software and/or hardware system. Typically, a benchmark harness contains components that generate the payload data, execute the database workload, monitor and collect data, and even visualise the measured results [34].

For instance, the harness OLTPBench [11,13] is a popular [24] academic open source project [38] written in Java. At the time of writing, the harness implements 19 benchmarks, including the three benchmarks introduced above. At the time of writing, Google Scholar reports over 280 citations of the full article [13] and the project is rated with over 330 stars on GitHub [38] and almost 250 forks.

2.3 JVM and Garbage Collectors

The Java Platform is the specification of a programming language and runtime environment and libraries, which is implemented by various Java Virtual Machines (JVMs). The open source implementation OpenJDK [40] has been used as reference implementation since Java Version 7 [41].

The *Java Virtual Machine* (JVM) is responsible for all aspects of executing a Java application, including memory management and communication with the operating system. The JVM is also a specification [31], with different implementations. The two most common implementations are the HotSpot JVM [40] by OpenJDK and the OpenJ9 JVM [39], an implementation maintained by the Eclipse Foundation.

The JVM has the concept of safepoints. While implementations differ between JVMs, in general, an executing thread is in a *safepoint* when its state is well

described. Operations such as executing Java Native Interface (JNI) code require a local safepoint, whereas others, such as certain garbage collection operations require a global safepoint. A global safepoint is a *Stop-The-World (STW)* pause, as all threads have to reach a safepoint and do not proceed until the JVM so decides. The latency of a STW pause is the time from the first thread reaches its safepoint until the last thread reaches its safepoint plus the time of performing the operation that requires STW.

Java is a garbage-collected language. The *garbage collector* (GC) is an integrated component of any JVM and responsible for managing the heap memory, *i.e.*, removing unused objects [31]. These housekeeping tasks can follow different strategies, that target different optimisation targets, such as optimising for low latency. The GC is configured at JVM startup, and additional tuneables can be applied to both, the JVM and the GC of choice. This allows for optimising a Java applicatino for peak performance based on its environmental conditions, such as hardware aspects or the specific area of use. However, most GC implementations [7, 8, 15, 42, 45] require a global safepoint during their collection phases, which introduces indeterministic latencies. Azul's C4 GC [9, 46] overcomes the issue of STW pauses by exploiting read barriers and virtual memory operations, provided by custom hardware or a Linux kernel module, for sustained pauseless garbage collection.

3 Experiments

In the following, we report on the results of our experiments with OLTPBench. As a baseline, we execute a minimal database workload (with the NoOp benchmark), while de-facto disabling the garbage collector (using the HotSpot JVM configured with the Epsilon garbage collector). This is designed to reveal additional latencies imposed by the benchmark harness. We further configure the harness with special-purpose garbage collectors designed for different scenarios, *e.g.*, which cause only low latencies, and contrast this with the default garbage collectors.

Experimental Setup. All experiments are performed with OLTPBench, executing the built-in NoOp, TPC-C and YCSB against PostgreSQL and MariaDB. For NUMA awareness, the database server processes as well as the benchmark processes are pinned to CPUs of the same NUMA node.

Benchmark Configuration. Each benchmark is configured with a 10-second warmup phase, to warmup the database caches, buffer pools etc., followed by a 60-second measurement phase. The isolation level is set to serialisable and the requests are sent in a closed-loop fashion (a new request will not be sent until the response of the previous request has been received and processed). The requests are issued by ten worker threads, *i.e.*, ten parallel connections.

TPC-C is configured with a scale factor of ten, resulting in ten independent warehouses. Each transaction relates to a specific warehouse using the warehouse ID (primary key). The warehouse IDs are distributed uniformly over all available

worker threads, hence each worker thread executes transactions only on its dedicated warehouse. This leads to a distribution of one worker per warehouse. OLTPBench implemented TPC-C as "good faith" and therefore deviates from the TPC-C specification in some places [13][4].

For YCSB[5], we use a scale factor of 1,200, resulting in a table with 1.2 million records. The workload distribution is as follows: 50% read, 5% insert, 15% scan, 10% update, 10% delete and 10% read-modify-write transactions, while all but scan access a single record based on the primary key. The primary key selection is based on a Zipfian distribution. All these values are default settings provided by OLTPBench.

In the current implementation of OLTPBench, the NoOp benchmark is only supported by PostgreSQL. In case of an empty query, PostgreSQL will acknowledge the query and report successful execution. MariaDB instead reports an empty query error, which results in a Java runtime exception on benchmark side, which, in turn, results in different code paths, compared with PostgreSQL. To promote comparable behavior, we enhanced both, MariaDB and OLTPBench: In OLTPBench, we disabled explicit *commit*s after transactions[6]. Additionally, we enhanced MariaDB to interprete the empty statement '`;`' internally as a comment (`--`), *i.e.*, as a NoOp.

Java and GC Settings. To run OLTPBench, we use Java version 16.0.1 (OpenJDK). As JVMs, we measure with both, the HotSpot JVM and once OpenJ9 JVM. For the HotSpot JVM, we used the garbage collectors G1 (default) [42], Shenandoah [7], ZGC [8] and Epsilon [45], a pseudo garbage collector, that leaves all objects in memory and performs no garbage collection at all. For OpenJ9, we used gencon (default) [15] and metronome [15]. Table 1 provides an overview about the strategies of the used GCs. We chose the default GC for HotSpot JVM and OpenJ9 JVM as they are probably the starting point for most measurements done with a Java application. In addition, we choose all low latency GC strategies with low STW pauses for precise latency measurements.

Experiment Execution. OLTPBench sets per default a maximum heap size of 8 GiB (JVM option `-Xmx8G`), which we also used for our experiments, with the exception of Epsilon GC. As the Epsilon GC does not perform garbage collection, we need to enlarge the heap size to keep a large amount of objects. In total, the Epsilon GC requires 180 GiB of heap space available of which 160 GiB were pre-allocated upon startup. During the 60-second measurement, the 160 GiB were sufficient and therefore no latencies were introduced due to an increase of the heap. The 20 GB buffer for creating the result files was partly needed by OLTPBench.

RR: Als Leser würde ich eine Erklärung für die 8 GB erwarten. Warum nicht 6.5GB?

[4] For example, TPC-C defines ten terminals (workers) per warehouse and each customer runs through a thinking time at one terminal, which is also eliminated by OLTPBench.

[5] Caused by a runtime error with Java versions ≥ 9, we had to increase the versions of the *jaxb-api* and *jaxb-impl* dependencies used by OLTPBench from 2.3.0 to 2.3.1.

[6] A commit after an empty query does not have any effects on execution

In order to log latencies introduced by the JVM, we use the unified logging mechanism, introduced in HotSpot JVM for Java 9 [28]. HotSpot JVM allows for logging all safepoint events, including the garbage collection events. OpenJ9 JVM also supports unified logging, but only supports garbage collection events and thus no safepoint events [14]. Here, we log the events reported by the GC and use these latencies as overhead from the JVM. We refer to Section 5 for a discussion of this tradeoff.

Table 1. Garbage collectors for the HotSpot JVM and OpenJ9 JVM with their area of specialisation.

JVM	GC	Design
HotSpot	G1	Balance between throughput and latency.
	Z	Low latency.
	Shenandoah	Low latency.
	Epsilon	Experimental setting. No GC tasks are performed, except for increasing heap.
J9	gencon	Transactional applications with short-lived objects.
	metronome	Low latency.

RR: Verwenden wir das Feature nicht, um zu beweisen, dass unsere Events wirklich aus dem GC kommen? Ich würde schreiben, dass das lediglich ein Zusatz ist, den wir verwenden, um unsere Hypothese zu bestätigen. Andernfalls kann man uns angreifen und sagen: Naja, dieses Logging macht ja auch wieder Latenz.

Execution Platform. All measurements are performed on a Dell PowerEdge R640, with two Intel Gold 6248R CPUs (24 cores per CPU, 3.0 GHz) and 384 GB of main memory. To avoid distortions from CPU frequency transitions, we disable Intel®™ Turbo Boost®™, and operate all cores in the performance P-State for a permanent core frequency of 3.0 GHz. Because of resource contention, simultaneous multithreading (SMT) is known to cause undesired side-effects on low-latency real-time systems [32]. Hence, we disable SMT.

To avoid cross-NUMA effects, OLTPBench as well as database server processes execute on 22 cores of the same NUMA node: OLTPbench and the database server (either MariaDB or PostgreSQL) exclusively execute on 11 cores, respectively. This ensures that one worker of the benchmark, which is pinned to a dedicated CPU, is connected to one worker of the database, which is pinned to a dedicated CPU as well. The remaining two cores of the NUMA node are reserved for remaining processes of the system.

The server runs Arch Linux with Kernel version 5.12.5. The benchmark as well as the database server are compiled from sources. For OLTPBench, we use git hash #6e8c04f, for PostgreSQL Version 13.3, git hash #272d82ec6f and for MariaDB Version 10.6, git hash #609e8e38bb0.

3.1 Results

As mentioned before, we evaluate MariaDB and PostgreSQL. Our evaluation shows that the results are virtually independent of the DBMS. Hence, and due to the limited size of the article, we present the results for MariaDB. For PostgreSQL, we refer to our supplementary website for the detailed results with PostgreSQL[7]. For the final paper, we will also provide a full reproducibility package containing all our scripts and measured data for a complete reproducibility of our experiments.

[7] https://github.com/sdbs-uni-p/tpctc2021

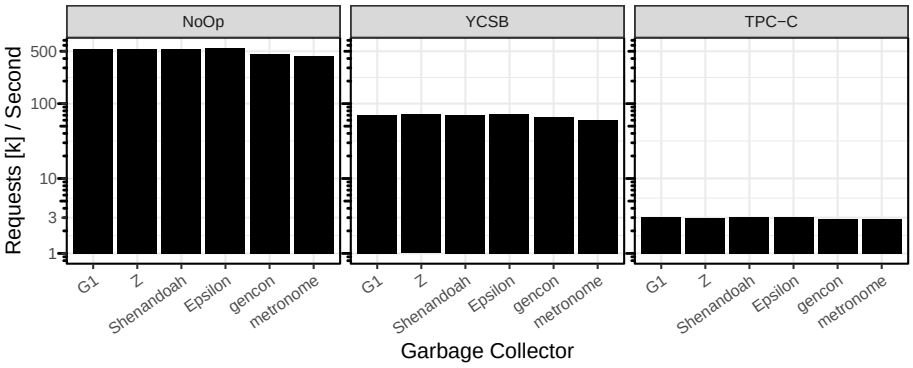

Fig. 1. Database throughput for MariaDB, in thousand requests per second for benchmarks NoOp, YCSB and TPC-C, and different JVM/GC configurations of OLTPBench. Throughput is affected marginally by the choice of JVM, but not the GC.

We follow a top-down approach: We first measure latencies on the level of single transations, under variation of different JVM and GC configurations with the NoOp, YCSB, and TPC-C benchmarks. We then characterise the latency distributions, and systematically investigate on the latency long tail.

Throughput. Figure 1 shows the throughput measured in thousand requests per second, for different benchmarks, JVMs, and GCs. For TPC-C, throughput is commonly reported as NewOrder transactions per minute (tpmC), but we deviate for a better comparability between the different benchmarks.

For all three benchmarks, Figure 1 reports a difference in performance between the two JVMs: Compared to OpenJ9 JVM, HotSpot JVM has about 17% – 28% more throughput for the NoOp benchmark, whereas it is about 5% – 18% for YCSB and only about 5% more for the TPC-C benchmark.

Naturally, the choice of JVM for OLTPBench has a stronger influence for benchmarks with in which comparatively little time is spent on the database side. To put this in context: By executing the NoOp benchmark with about 500K requests per second, we spend much more time in the process of OLTPBench compared to the TPC-C benchmark with only 3K requests per second.

Latency distribution. The distribution of latencies reported by OLTPBench is characterised as boxplots in Figure 2.

Overall, there is little variation in the median latencies within a benchmark. Comparing the median latencies of the two JVMs for the NoOp benchmark, we see a lower median latency for HotSpot JVM than for OpenJ9 JVM. This is a mere difference of 0.003 ms or 0.004 ms, based on about 500k requests per second. As the NoOp benchmark generates only a small load on the database side, the maximum latencies reported are candidates for latencies introduced by the Java

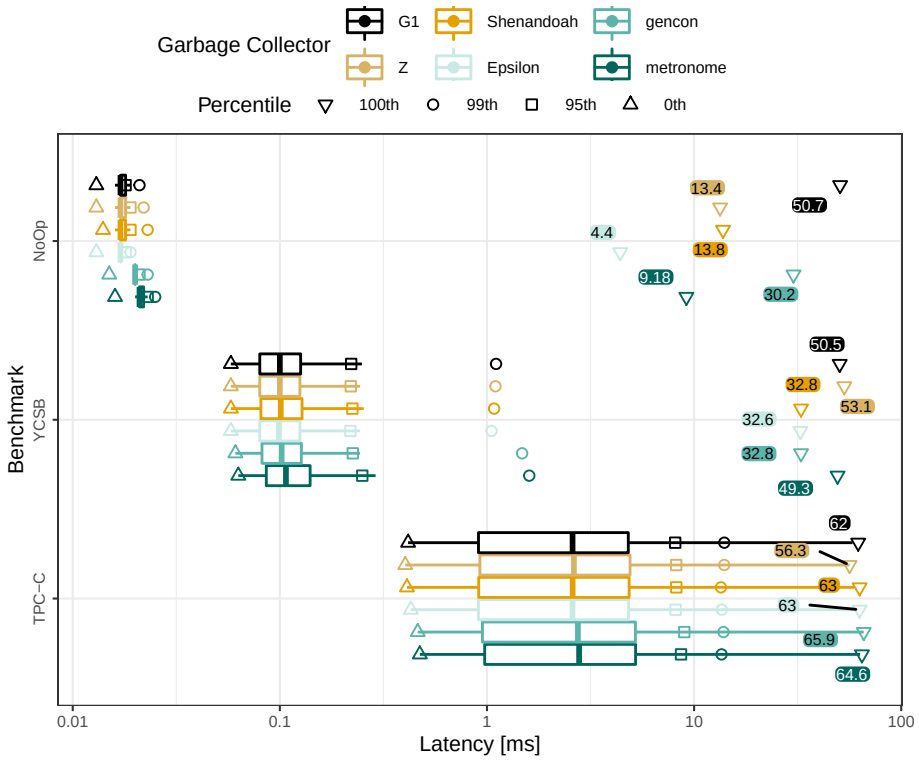

Fig. 2. The latency distributions measured by OLTPBench for three benchmarks, visualised as box plots. Key percentiles are highlighted.

environment of the benchmark harness. As can be expected, Epsilon GC has the lowest maximum latency for the NoOp benchmark.

The minimum (0th percentile) and maximum (100th percentile) latencies, as well as the 95th and 99th latency percentiles, are separately marked. The absolute value of the maximum latency is also labeled.

The YCSB benchmark shows strong variance in maximum latencies, depending on the garbage collector used. For GCs G1, Z and gencon, a maximum latency of around 50 ms is recorded, whereas the other GCs display a maximum latency of about 30 ms. We inspect these latencies more closely in the following. Nevertheless, the distribution of the latencies is rather uniform for all six garbage collectors, except the 99th percentile latency, where OpenJ9 latencies exceed that of HotSpot.

The different JVM/GC configurations result in near-identical latency distributions for the TPC-C benchmark: Due to the larger share of time spent on the database side (compared to the other benchmarks), the latencies introduced on side of the benchmark harness do not weigh in as much in comparison.

Latency time series. Figures 3 through 5 show time series plots for the benchmarks NoOp, YCSB, and TPC-C. Red, labeled triangles mark minimum and maximum latencies, as observed by OLTPBench. In order to prevent overplotting, we sampled the latencies except for extreme values. Ochre dots represent sampled, normal observations. A latency is considered an extreme value and displayed as grey dot, if it is outside a pre-defined range. We define benchmark-specific sampling rates and extreme value ranges, as detailed further below.

We extract latencies from the JVM logs and superimpose them as black dots. In benchmarks with mixed, randomised workloads, we cannot associate a given JVM latency with a specific query, so we visualise all JVM latencies. The red line shows a sliding mean window computed over 1,000 measuring points to make latency fluctuations more visible.

The time series plots for the YCSB and TPC-C benchmark are provided for selected queries only. We refer to our online supplement (see Footnote 7) for the full set of charts, which, as our readers will notice, do not reveal new insights. Similarly, we do not visualise the Shenandoah GC as it behaves similar to the Z GC, and metronome GC, with a similar latency pattern as gencon GC.

NoOp Benchmark. The latency time series for the NoOp benchmark is shown in Figure 3. To avoid overplotting, we used a sampling rate of 0.0001% for standard values, i.e., values in between the 0.025th and 99.975th percentile.

@Wolfgang, is it "statistically safe" to phrase ist like this?

OLTPBench, executed with the Epsilon GC, shows that this setup has the lowest latency possible, the JVM is only active for a short time at the very end. This measurement shows us that regular outliers appear at about 1 ms and can be used as a reference for comparison with other GCs. The measurement of the GC G1 shows us that the GC causes the maximum latency or the tail of the latencies. Almost each latency higher than 10 ms was introduced by the GC because the latencies reported by OLTPBench and these reported by the JVM show the same pattern and match each other.

YCSB Benchmark. We show the time series latency of the YCSB benchmark in Figure 4. We report the latencies measured by the G1, Z, Epsilon and metronome GC and selected the two transaction types ReadRecord (read transaction) and UpdateRecord (write transaction). We used a sampling rate of 0.05% for standard values and the 99.975th and 0.025th latency percentile are the limits for a latency to be marked as extreme value.

The Epsilon GC is again the reference and except for the maximum latency, all tail latencies are within the range of 1 ms and 5 ms. By comparing the ReadRecord latencies, again the G1 GC is responsible for the tail latencies occurring in this transaction. The write transaction shows a similar behaviour, but here outliers on the database side are responsible for the maximum latency, nevertheless again the G1 GC latency defines the tail.

TPC-C Benchmark. The time series latency of TPC-C is given in Figure 5. The sampling rate of standard values of the NewOrder transaction is set to 0.05%

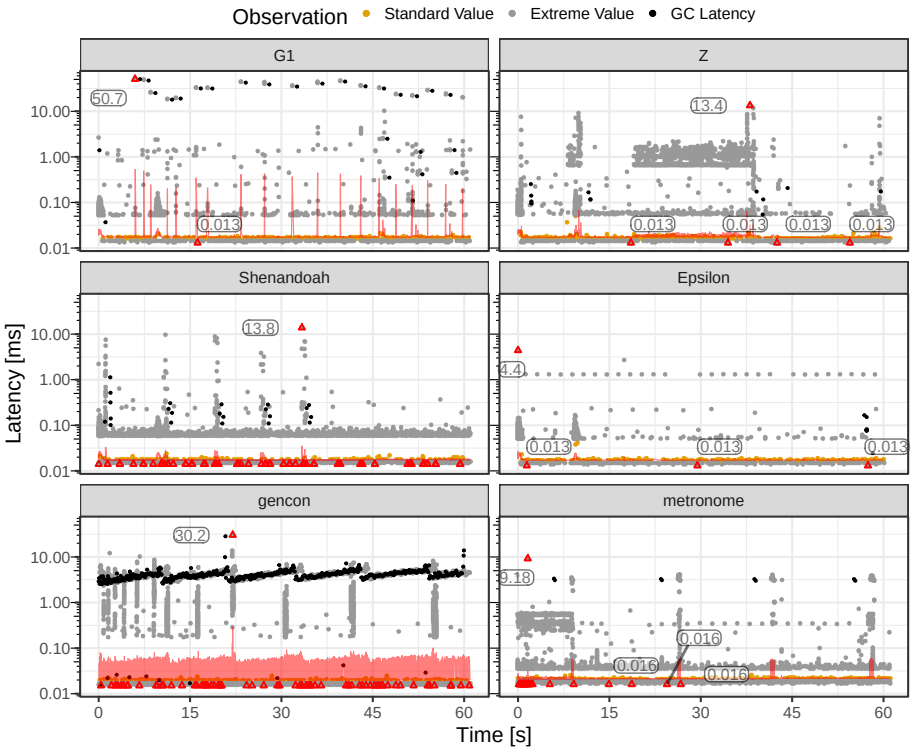

Fig. 3. Latency time series of the NoOp benchmark. The minimum and maximum latencies measured with OLTPBench are marked by red, labelled triangles. Grey dots are extreme values. Ochre dots are standard observations. The latencies from the JVM log file are superimposed in black. The red line shows the sliding mean window.

and for OrderStatus to 5%. Extreme values are marked as such if they exceed the 99.75th percentile or subceed the 0.25th percentile.

For this particular benchmark, we cannot identify a clear influence of the JVM. The transactions, especially the write transactions, are so heavyweight that most of the time is spent on the database. This does not leave much time to be spent on the benchmark side, which is be influenced by Java. Furthermore, due to the low number of requests per second (about 3k), there has not be performed much garbage collection as for the other benchmarks because not that many Java objects are created which have to be cleaned up afterwards. The same applies to the read transaction.

4 Discussion

Our experiments show that in a popular Java-based benchmark harness, the choice of the Java environment, that is, the JVM and its GC strategy, impacts

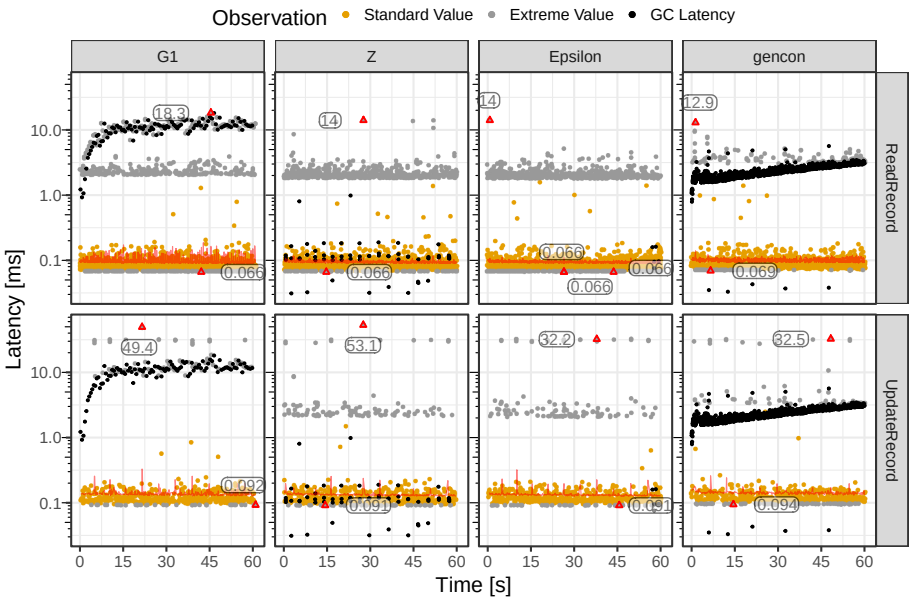

Fig. 4. Latency time series of the YCSB benchmark for a read (ReadRecord) and a write (UpdateRecord) transactions. Labels and colors as in Figure 3.

(tail) latency measurements. By super-imposing the latencies extracted from JVM log files on the latency time series reported by TPCBench, we can make this connection visually apparent. By setting up a baseline experiment, with garbage collection de-facto disabled, and a minimalist database workload, we can successfully quantify the imposed latencies.

Naturally, for lightweight database workloads (in our experiments, YCSB), this non-productive overhead is more noticeable in relation to the actual query execution time. We can show that different GC strategies translate to different patterns in the tail latencies.

Interestingly, while researchers and practitioners optimise latencies in the realm of micro seconds [2], the latencies imposed by the benchmark harness reach the ballpark of milliseconds. Evidently, this factor of one thousand proves these effects are non-negligible, and deserve further study.

Our observations are replicable for PostgreSQL. We provide the data and full set of plots on our supplementary website, along with a reproduction package.

5 Threats to Validity

We applied great care and diligence in analysing the garbage collector logs. Yet as the logging mechanisms differ between JVMs, we must deal with some level of uncertainty: The HotSpot JVM logs all safepoints (including, but not restricted

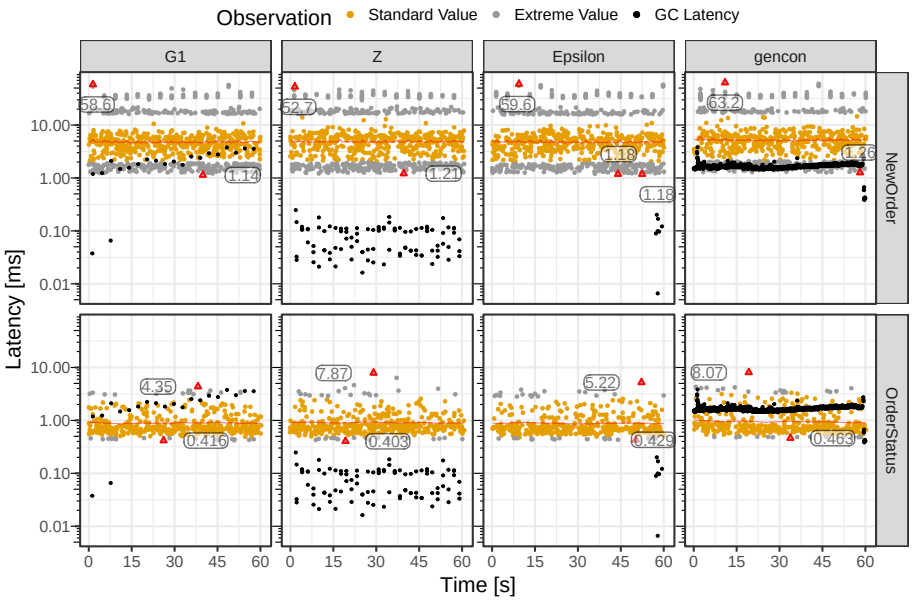

Fig. 5. Latency time series of the TPC-C benchmark for a read (OrderStatus) and a write (NewOrder) transactions. Labels and colours as in Figure 3.

to garbage collection events), whereas the OpenJ9 JVM logs only the the garbage collection events. As the GC events dominate the logged safepoint events, we treat the reported latencies in both logs uniformly. In addition, we do not distinguish between local and global safepoints, as local safepoints can also block a thread.

Further, the latencies reported by OLTPBench and the latencies logged by the JVM are two distinct sources of data. As usual, data integration brings about uncertainties, so the time series plots might be shifted minimally. In summary, we consider the above threats to the validity of our results as minor.

One further threat to validity is that we only focus on the Java environment of the benchmark harness, but no other possible sources of systemic noise (such as caused by the hardware). We have diligently configured the execution platform to reduce sources of noise (*e.g.*, having disabled SMT). Moreover systemic noise is typically in the range of micro seconds [2]. Since the harness-induced latencies are in the millisecond range (exceeding them by a factor of one thousand, and clearly traceable back to the harness), we may dismiss this threat.

6 Related Work

Databases, as core component of data-intensive systems, are important contributors to system-wide tail latency effects. Likewise, they have started to enjoy increasing popularity in real-time scenarios like sensor networks or IoT [17, 43],

where the most crucial service indicator is determinism and *maximum* latency, instead of average-case performance. Care in controlling excessive latencies must be exercised in all cases.

Several measures have been suggested to this end, and address software layers above, inside and below the DBMS. For instance, optimising end-to-end communications [19] tries to alleviate the issue from above. Specially crafted real-time databases [1], novel scheduling algorithms in scheduling/routing queries [25, 26,44], transactional concepts [6], or query evaluation strategies [20,49], work from inside the database. Careful tailoring of the whole software stack from OS kernel to DB engine [30, 33], or crafting dedicated operating systems [5, 21–23, 35, 36] to leverage the advantages of modern hardware in database system engineering (*e.g.*, [16, 29]), contribute to solutions from below the database.

In our experiments, we execute the well-established YCSB and TPC-C benchmarks, which are supported by the OLTPBench harness out-of-the-box. However, further special-purpose benchmarks have been proposed specifically for measuring tail latencies, as they arise in embedded databases on mobile devices [37]. This contribution is related to our work insofar as the authors also unveil sources of measurement error, however, errors that arise when measuring the performance of embedded databases at low throughput.

There are further benchmark harnesses from the systems research community that specifically target tail latencies. These capture latencies across the entire systems stack, while in the database research community, we benchmark the database layer in isolation. Harnesses such as TailBench [27] or TreadMill [50] define such complex application scenarios (some involving a database layer).

In its full generality, the challenge of benchmarking Java applications, including jitter introduced by garbage collection, is discussed in [18]. We follow the best practices recommended in this article, as we classify outliers as either systematic or probabilistic. We present the latency distributions clearly, and have carried out sufficiently many test runs. Different from the authors of [18], we do not apply hypothesis tests, since we are mostly interested in latencies maxima, rather than confidence in averaged values.

It has been reported that database-internal garbage collection [3, 29] can also cause latency spikes. Yet in our work, we consider the effects of garbage collection inside the test harness, rather than the database engine.

7 Conclusion and Outlook

Tail latencies in database query processing can manifest as acute pain points. As a prerequisite to addressing possible causes in database system engineering, we need to be able to faithfully measure these latencies in the first place. This article shows that Java-based benchmark harnesses serve well for measuring database throughput, or 95th or 99th percentile latencies. However, we show that intrinsic aspects of the programming language, Java in particular, have significant impact on the measurement when it comes to capturing tail latencies: The choice of JVM and garbage collector in the benchmark harness is a non-negligible source

of undeterministic noise. Specifically, for database workloads composed of low-latency queries (*e.g.*, as in the YCSB benchmark), we risk distorted measurements which an lead us to chasing down ghosts in database systems engineering.

We conclude that in our scenario, we need to reconsider the toolsuites and their configuration.

RR: Hmm. Würde eher schreiben, das man idealerweise Sprachen verwenden sollte, die minimale bzw. keine inhärenten Seiteneffekte während der Ausführung haben, und dass dafür Java, schlichtweg, ungeeignet ist. Ruhig scharf formulieren. Weiterhin kann man hier vorsichtig teasen, dass sie da die future work hinbewegen wird.

References

1. Real-Time Database Systems: Architecture and Techniques, vol. 593 (2001)
2. Barroso, L., Marty, M., Patterson, D., Ranganathan, P.: Attack of the killer microseconds. Commun. ACM **60**(4), 48–54 (Mar 2017), https://doi.org/10.1145/3015146
3. Böttcher, J., Leis, V., Neumann, T., Kemper, A.: Scalable Garbage Collection for In-Memory MVCC Systems. Proceedings of the VLDB Endowment **13**(2), 128–141 (Oct 2019). https://doi.org/10.14778/3364324.3364328
4. Brutlag, J.: Speed Matters for Google Web Search. Available at: https://venturebeat.com/wp-content/uploads/2009/11/delayexp.pdf (Jun 2009), blogpost
5. Cafarella, M.J., DeWitt, D.J., Gadepally, V., Kepner, J., Kozyrakis, C., Kraska, T., Stonebraker, M., Zaharia, M.: DBOS: A Proposal for a Data-Centric Operating System. CoRR **abs/2007.11112** (Jul 2020), https://arxiv.org/abs/2007.11112
6. Chen, X., Song, H., Jiang, J., Ruan, C., Li, C., Wang, S., Zhang, G., Cheng, R., Cui, H.: Achieving Low Tail-Latency and High Scalability for Serializable Transactions in Edge Computing. In: Proceedings of the Sixteenth European Conference on Computer Systems. p. 210–227. EuroSys '21 (2021). https://doi.org/10.1145/3447786.3456238
7. Clark, I.: Shenandoah GC. Available at: https://wiki.openjdk.java.net/display/shenandoah (Apr 2021), v. 138. Accessed on June 13, 2021
8. Clark, I.: ZGC. Available at: https://wiki.openjdk.java.net/display/zgc (Mar 2021), v. 124. Accessed on June 13, 2021
9. Click, C., Tene, G., Wolf, M.: The pauseless GC algorithm. In: Proceedings of the 1st International Conference on Virtual Execution Environments. pp. 46–56. VEE '05 (2005). https://doi.org/10.1145/1064979.1064988
10. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. pp. 143–154. SoCC '10 (2010). https://doi.org/10.1145/1807128.1807152
11. Curino, C., Difallah, D.E., Pavlo, A., Cudré-Mauroux, P.: Benchmarking OLTP/web databases in the cloud: the OLTP-bench framework. In: Proceedings of the Fourth International Workshop on Cloud Data Management. pp. 17–20. CloudDB '12 (2012). https://doi.org/10.1145/2390021.2390025
12. Dean, J., Barroso, L.A.: The Tail at Scale. Communications of the ACM **56**(2), 74–80 (Feb 2013). https://doi.org/10.1145/2408776.2408794
13. Difallah, D.E., Pavlo, A., Curino, C., Cudré-Mauroux, P.: OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. Proceedings of the VLDB Endowment **7**(4), 277–288 (Dec 2013). https://doi.org/10.14778/2732240.2732246
14. Eclipse Foundation: -Xlog. Available at: https://www.eclipse.org/openj9/docs/xlog/
15. Eclipse Foundation: Garbage collection policies. Available at: https://www.eclipse.org/openj9/docs/gc/, accessed on June 13, 2021

16. Fent, P., van Renen, A., Kipf, A., Leis, V., Neumann, T., Kemper, A.: Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In: 36th IEEE International Conference on Data Engineering. pp. 1477–1488 (2020). https://doi.org/10.1109/ICDE48307.2020.00131
17. Garcia-Arellano, C., Roumani, H., Sidle, R., Tiefenbach, J., Rakopoulos, K., Sayyid, I., Storm, A., Barber, R., Ozcan, F., Zilio, D., et al.: Db2 event store: a purpose-built IoT database engine. Proceedings of the VLDB Endowment **13**(12), 3299–3312 (2020)
18. Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. SIGPLAN Not. **42**(10), 57–76 (Oct 2007). https://doi.org/10.1145/1297105.1297033
19. Gessert, F.: Low Latency for Cloud Data Management. Ph.D. thesis, University of Hamburg, Germany (2019), http://ediss.sub.uni-hamburg.de/volltexte/2019/9541/
20. Giannikis, G., Alonso, G., Kossmann, D.: SharedDB: Killing One Thousand Queries with One Stone. Proceedings of the VLDB Endowment **5**(6), 526–537 (Feb 2012). https://doi.org/10.14778/2168651.2168654
21. Giceva, J.: Operating System Support for Data Management on Modern Hardware. IEEE Data Engineering Bulletin **42**(1), 36–48 (2019), http://sites.computer.org/debull/A19mar/p36.pdf
22. Giceva, J., Salomie, T.I., Schüpbach, A., Alonso, G., Roscoe, T.: COD: Database / Operating System Co-Design. In: Sixth Biennial Conference on Innovative Data Systems Research, Online Proceedings. CIDR'13 (2013), http://cidrdb.org/cidr2013/Papers/CIDR13_Paper71.pdf
23. Giceva, J., Zellweger, G., Alonso, G., Roscoe, T.: Customized OS support for data-processing. DaMoN pp. 2:1–2:6 (Jun 2016). https://doi.org/10.1145/2933349.2933351
24. Hofmann, G., Riehle, D., Kolassa, C., Mauerer, W.: A Dual Model of Open Source License Growth. In: Open Source Systems Conference (OSS'13). IFIP AICT, vol. 404, pp. 245–256. Springer (2013), http://www.se-rwth.de/publications/A-Dual-Model-of-Open-Source-License.pdf
25. Jaiman, V., Mokhtar, S.B., Quéma, V., Chen, L.Y., Riviere, E.: Héron: Taming Tail Latencies in Key-Value Stores Under Heterogeneous Workloads. In: 37th IEEE Symposium on Reliable Distributed Systems. pp. 191–200 (2018). https://doi.org/10.1109/SRDS.2018.00030
26. Jaiman, V., Mokhtar, S.B., Rivière, E.: TailX: Scheduling Heterogeneous Multiget Queries to Improve Tail Latencies in Key-Value Stores. In: Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science, vol. 12135, pp. 73–92 (2020). https://doi.org/10.1007/978-3-030-50323-9_5
27. Kasture, H., Sanchez, D.: Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In: 2016 IEEE International Symposium on Workload Characterization (IISWC). pp. 1–10 (2016). https://doi.org/10.1109/IISWC.2016.7581261
28. Larsen, S., Arvidsson, F., Larsson, M.: JEP 158: Unified JVM Logging. Available at: https://openjdk.java.net/jeps/158, accessed on June 13, 2021
29. Lersch, L., Schreter, I., Oukid, I., Lehner, W.: Enabling Low Tail Latency on Multicore Key-Value Stores. Proceedings of the VLDB Endowment **13**(7), 1091–1104 (Mar). https://doi.org/10.14778/3384345.3384356
30. Li, J., Sharma, N.K., Ports, D.R.K., Gribble, S.D.: Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In: Proceedings of the ACM Symposium on Cloud Computing. p. 1–14. SOCC '14 (2014). https://doi.org/10.1145/2670979.2670988

31. Lindholm, T., Yellin, F., Bracha, G., Buckley, A., Smith, D.: The Java Virtual Machine Specification - Java SE 16 Edition. Available at: https://docs.oracle.com/javase/specs/jvms/se16/jvms16.pdf (2 2021), accessed on June 13, 2021
32. Mauerer, W.: Professional Linux Kernel Architecture. John Wiley & Sons (2010)
33. Mauerer, W., Ramsauer, R., Filho, E.R.L., Lohmann, D., Scherzinger, S.: Silentium! Run-Analyse-Eradicate the Noise out of the DB/OS Stack. In: Datenbanksysteme für Business, Technologie und Web (BTW). LNI, vol. P-311, pp. 397–421 (2021). https://doi.org/10.18420/btw2021-21
34. Michael, N.: Benchmark Harness. In: Encyclopedia of Big Data Technologies. pp. 137–141 (2019). https://doi.org/10.1007/978-3-319-63962-8_134
35. Mühlig, J., Müller, M., Spincyk, O., Teubner, J.: MxKernel: A Novel System Software Stack for Data Processing on Modern Hardware. Datenbank-Spektrum **20**(3), 223–230 (Nov 2020). https://doi.org/10.1007/s13222-020-00357-5
36. Müller, M., Spinczyk, O.: MxKernel: Rethinking Operating System Architecture for Many-core Hardware. In: 9th Workshop on Systems for Multi-core and Heterogenous Architectures (2019)
37. Nuessle, C., Kennedy, O., Ziarek, L.: Benchmarking Pocket-Scale Databases. In: Performance Evaluation and Benchmarking for the Era of Cloud(s). Lecture Notes in Computer Science, vol. 12257, pp. 99–115 (2019). https://doi.org/10.1007/978-3-030-55024-0_7
38. OLTPBenchmark.com: OLTPBench. Available at: https://github.com/oltpbenchmark/oltpbench, accessed on June 13, 2021
39. eclipse openj9: OpenJ9. Available at: https://github.com/eclipse-openj9/openj9, accessed on June 13, 2021
40. OpenJDK: JDK. Available at: https://github.com/openjdk/jdk, accessed on June 13, 2021
41. Oracle: Java Platform, Standard Edition 16 Reference Implementations. Available at: https://jdk.java.net/java-se-ri/16, accessed on June 13, 2021
42. Oracle: Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide - 9 Garbage-First Garbage Collector. Available at: https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm#JSGCT-GUID-ED3AB6D3-FD9B-4447-9EDF-983ED2F7A573 (Oct 2017), online documentation
43. Paparrizos, J., Liu, C., Barbarioli, B., Hwang, J., Edian, I., Elmore, A.J., Franklin, M.J., Krishnan, S.: VergeDB: A Database for IoT Analytics on Edge Devices. In: CIDR (2021)
44. Reda, W., Canini, M., Suresh, L., Kostić, D., Braithwaite, S.: Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In: Proceedings of the Twelfth European Conference on Computer Systems. p. 95–110. EuroSys '17 (2017). https://doi.org/10.1145/3064176.3064209
45. Shipilev, A.: JEP 318: Epsilon: A No-Op Garbage Collector (Experimental). Available at: https://openjdk.java.net/jeps/318 (Sep 2018), accessed on June 13, 2021
46. Tene, G., Iyengar, B., Wolf, M.: C4: the continuously concurrent compacting collector. In: Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011. pp. 79–88. ACM (2011), https://doi.org/10.1145/1993478.1993491
47. The Transaction Processing Council: TPC-C Benchmark (Revision 5.11). Available at: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf (2 2010), accessed on June 13, 2021

48. Transaction Processing Performance Council: TPC-Homepage. Available at: http://tpc.org/
49. Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., Kossmann, D.: Predictable Performance for Unpredictable Workloads. Proceedings of the VLDB Endowment **2**(1), 706–717 (Aug 2009). https://doi.org/10.14778/1687627.1687707
50. Zhang, Y., Meisner, D., Mars, J., Tang, L.: Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). pp. 456–468 (2016). https://doi.org/10.1109/ISCA.2016.47