Benchmarking Databases with Varying Value Sizes [Experiment, Analysis, and Benchmark]

Danushka Liyanage
University of Sydney
Sydney, Australia
danushka.liyanage@sydney.edu.au

Shubham Pandey
Independent Researcher
Austin, Texas, USA
shubhambeethoven@gmail.com

Joshua Goldstein
Texas A&M University
College Station, Texas, USA
jgoldstein345@gmail.com

Michael Cahill
University of Sydney
Sydney, Australia
michael.cahill@sydney.edu.au

Akon Dey
Predictable Labs, Inc.
Foster City, USA
akon.dey@gmail.com

Alan Fekete
University of Sydney
Sydney, Australia
alan.fekete@sydney.edu.au

Uwe Röhm
University of Sydney
Sydney, Australia
uwe.roehm@sydney.edu.au

ABSTRACT

The performance of database management systems (DBMS) is traditionally evaluated using benchmarks that focus on fixed-size relational database workloads. However, real-world workloads in key/value stores, document databases, and graph databases often exhibit significant variability in value sizes, which can lead to performance anomalies, particularly when popular records grow disproportionately large. Existing benchmarks fail to account for this variability, leaving an important aspect of DBMS behavior underexplored.

In this paper, we address this gap by extending the Yahoo! Cloud Serving Benchmark (YCSB) to include an "extend" operation, which appends data to record fields, simulating the growth of values over time. Using this modified benchmark, we evaluate the performance of three popular DBMS backends: MongoDB, MariaDB with the InnoDB storage engine, and MariaDB with the MyRocks storage engine. Our experiments alternate between extending values and executing read-only query workloads, revealing significant performance differences driven by storage engine design and their handling of variable-sized values.

Our contributions include the introduction of a novel benchmarking approach to evaluate the impact of growing value sizes, a comparative performance analysis of popular DBMS systems under this workload, and insights into the performance challenges posed by large, non-uniform values. These findings highlight the need for more representative benchmarks that capture the dynamic nature of real-world workloads, providing valuable guidance for both practitioners and researchers.

PVLDB Reference Format:
Danushka Liyanage, Shubham Pandey, Joshua Goldstein, Michael Cahill, Akon Dey, Alan Fekete, and Uwe Röhm. Benchmarking Databases with Varying Value Sizes
[Experiment, Analysis, and Benchmark]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dliyanage/YCSB-IVS.

1 INTRODUCTION

Performance evaluation of database management systems (DBMS) is a critical aid to understanding their behavior under various workloads, and the development of widely adopted benchmarks has been seen as a driver for improvements in DBMS implementation. Benchmarks choose data and workload characteristics that simplify real-world applications, but they aim to keep essential features that stress possible inefficiencies of implementation, for example, by having significant contention, requirements for sometimes rolling back, expectations to scale the amount of data along with the workload, etc. If some aspect of the DBMS is not exercised by benchmarks, our community understanding of those issues may be weakened, and there is also a risk that platform engineers may not pay proper attention to designing for this.

In this paper, we consider a gap in prior work on benchmarking: the neglect of cases where some data items (records in tables, documents in a document store, values in a key value score) get longer and longer as the system evolves, eventually becoming quite long (say megabyte size). This can happen particularly when an item has an attribute whose type is array and where the application uses this array to store information about the item's history. In that case, some operations of the application append extra information to that attribute; if the item is a popular one and the database lasts a long time, the item may grow very long. This situation has not been well covered in previous benchmarks. Some (e.g., YCSB) have fixed size for all the items; while TPC-C does include some variation in record

size (due to VARCHAR fields such as C_DATA), this variation is limited, with every record never longer than 1KB.

The existence in real applications of items that grow in size over time can impact the performance of the system, depending on choices made in the implementation of the storage layer. These situations can cause fragmentation of the data, or may lead to frequent re-arrangements if the storage uses an update-in-place approach. Some systems store deltas to optimize the case where only a small part of an item is modified in an update. If these design aspects are not exercised in a benchmark, we will not know how well the choices were made. Thus, the focus of this paper is to propose a benchmark that examines what happens when items have lengths that increase significantly over time.

When the data includes some items that are growing steadily, there are several consequential changes to the data, and we aim to disentangle the performance impact of these. One impact of some records growing longer through history is that at any given time, the data items in the database will vary in length among themselves. Another consequence is that the total volume of data in the database will grow. Our benchmark is designed to measure performance with a database where some items keep growing, but we also compare this with performance when the item sizes vary, but not as a result of a history of growth. Also, we look at performance of two baselines with the same total volume and items of uniform size: one baseline has many small records, and the other has fewer records, which are each (equally) long.

We propose a benchmark which is built on the framework provided by Yahoo! Cloud Serving Benchmark (YCSB), a widely used benchmarking tool originally for key/value stores, but also with bindings for a wide variety of platforms including SQL ones. While the original YCSB workload has fixed size items (by default, 1000 bytes exactly), which can be read, updated, etc, we introduce an "extend" operation that appends data to one or more fields of a record, causing the length to grow over time. In contrast to the original YCSB, where data is loaded initially, and then a single measured run executes a mix of read, update etc operations, our benchmark loads data, and then proceeds in a sequence of epochs, each of which performs some number of extend operations applied to keys randomly chosen according to some distribution, and then we measure the metrics (throughput, latency etc) for a read-only workload against the state at the end of the extension phase. As we run successive epochs, some of the data items become longer and longer, and the read-only measurements show the slowdown this causes. For comparison, we also measure for each epoch, the read performance on a database with the same logical state (and thus the same distribution of item lengths), but no history of increase. We report the performance metrics for each epoch, and also report on two baselines which execute the queries against two datasets with equal-sized items and the same total volume as the given epoch; one baseline has the same number of items as in our benchmark (and so each item has length which is the average length found after extensions), and in the other baseline, all records have the initial (unextended) length, and so there are many more items than in the benchmark execution. We have applied our benchmark to evaluate the performance of several popular DBMS platforms – MongoDB, MariaDB with the InnoDB storage engine, and MariaDB with the MyRocks storage engine, and we report on the outcomes in this paper.

Our findings reveal significant performance differences between systems, driven by their storage engines and handling of large, variable-sized values. For example, some systems demonstrate robust performance even as values grow, while others exhibit surprising slowdowns. These results underscore the importance of incorporating varying value sizes into benchmarking methodologies to uncover hidden performance characteristics and better inform DBMS design and optimization.

This paper makes the following contributions:

- **A novel benchmarking approach for databases/key-value stores:** We extend YCSB to simulate realistic scenarios where value sizes grow over time, reflecting issues that occur in real-world workload patterns, but have not been effectively benchmarked before.
- **Extensive evaluation of popular DBMS platforms under our benchmark:** We present detailed performance analyses of MongoDB, MariaDB with InnoDB, and MariaDB with MyRocks, highlighting the unique strengths and weaknesses of each system in the face of items that grow in length through time.
- **Insights into DBMS performance under variable value sizes:** Our study identifies key factors that influence performance in the presence of large and non-uniform values, offering guidance for practitioners and researchers.

In Section 2 we discuss in more detail some real-world cases where items grow in length substantially over time. In Section 4 we describe the detailed operation of our new benchmark. In Section 5 we report on the measurements our benchmark gives, when run on three varied platforms, and we discuss what these experiments reveal.

Through this work, we aim to broaden the scope of DBMS performance evaluation, expanding from the current focus on fixed-size data items, to include workloads where some items can grow to be quite large. In doing so, we contribute to a deeper understanding of how modern DBMS systems perform under conditions that are more closely aligned with real-world applications. The source of our benchmark, and data from our experiments, is provided at https://github.com/dliyanage/YCSB-IVS

2 MOTIVATION

In numerous applications, array-valued attributes are a common feature within data records, playing an essential role despite not fitting the strict criteria of the first normal form (1NF) in traditional relational database models. These array attributes enable rich representations of complex data relationships and historical changes within the records. Despite their departure from conventional normalization principles, many relational databases (RDBMS) and modern NoSQL databases, such as document stores and graph databases, offer support for storing such multi-valued fields.

In practical applications, these array-type fields often serve critical functions. For example, an e-commerce platform might maintain a product record, complete with an array field that lists all previous prices at which the product was marketed. Each time the product is listed with a new price, the list of prices within the array attribute

expands with the addition of the latest offered price. Similarly, a social media site keeps track of posts and tags. Over time, additional tags are appended to the existing list of tags for each post.

In long-term database usage, array-valued fields in popular items can experience significant growth, reaching megabytes in length, while entries in less popular items may only reach kilobyte ranges. This stark divergence in item size within a single collection or table is characteristic of data that has grown incrementally over time. The potential for such wide variations in record size can have implications for database performance. It underscores the need to understand and benchmark how the gradual expansion of array fields influences the overall performance of the DBMS.

We describe real-world situations that were found in earlier industrial experience of some of the authors.

Record growth in Dgraph. Dgraph [11] is a platform that provides direct support for a data model as a graph. It is targeted at distributed systems of large scale with high update rates. Data storage patterns in Dgraph can lead to a disproportionate growth of values associated with a small subset of keys. This issue became particularly evident at Dgraph Labs, where it contributed to extended support calls, production outages at high-profile customer sites, and numerous sleepless nights.

Dgraph relies on the Badger key-value store for data persistence. Some keys, related to type information, gradually accumulate a large number of identifiers (*uids*) referencing records of that particular type [21]. As the database matures and contains more data, the size of these values scales in proportion to the overall growth of the database.

Compounding the issue, record deletions provide little relief due to the underlying storage mechanisms. Even with garbage collection in place, large record sizes remain a persistent challenge, exacerbating storage inefficiencies and operational complexity over time.

Document growth in MongoDB. MongoDB is a platform that directly supports a document data model. Certain application patterns with MongoDB can lead to documents expanding over time, such as maintaining an exhaustive history of events or tracking changes to a value across multiple iterations. In these scenarios, what begins as small increments to a document's size can compound over time, leading to substantial increases in document size after numerous updates.

This growth pattern is particularly concerning when each update causes the document to swell incrementally; without careful design, such expansion can precipitate memory and storage fragmentation. MongoDB addresses this challenge by delaying when small updates to large documents are persisted to storage [1]. Instead, they are cached and consolidated into fewer and larger updates upon subsequent write operations. This strategy mitigates the quadratic increase in work associated with handling numerous tiny updates and reduces the incidence of fragmentation.

The impact of this design choice is twofold: first, it alleviates the memory and storage overhead that would otherwise result from frequent small updates. Second, certain workloads experience considerable performance enhancement due to this optimization.

[1] https://www.mongodb.com/community/forums/t/in-place-partial-updates/240487

3 RELATED WORK

Databases Value/Object Sizes. The size of values or objects in databases plays a crucial role in query execution, as larger sizes impose greater processing demands, thereby slowing down performance. Consequently, object size has been a central consideration in numerous query optimization cost models [12, 15], often serving as a critical factor in addressing scalability challenges. For instance, Shapiro et al. [26] evaluated the performance of databases utilizing Binary Large Objects (BLOBs) and observed that increases in both database size and average object size significantly degrade transaction efficiency, highlighting a key limitation in database scalability.

It is evident that database developers have long recognized the trade-offs associated with value sizes, particularly the efficiency gains achieved by keeping value sizes small. This understanding has made data *compression* a fundamental feature in modern databases and key-value stores [19, 20, 30]. A quantitative assessment by Hurson et al. [18] in the early 1990s demonstrated that smaller object sizes significantly enhance the efficiency of object clustering, reducing I/O operations and achieving cost improvements ranging from a factor of 2 to 15.

While efforts are often made to keep value or object sizes as small as possible, there are numerous scenarios where value sizes naturally grow over time. Pelekis et al. [24] highlighted that size plays a critical role in spatio-temporal database models, especially in databases that undergo continuous changes.

Database Benchmarking. The 'Wisconsin' benchmarks [5], introduced in the early 1980s, were among the first systematic attempts to benchmark relational database systems, evaluating platforms like DIRECT, INGRES, ORACLE, and IDM500. Soon after, the TP1 benchmark (1985) emerged, measuring transaction throughput through components such as *transaction (DebitCredit)*, *OS and I/O (Sort)*, and *file system (Copy)*. Jim Gray's benchmarking handbook [16] formalizes database benchmarking practices, leading to the expansion of benchmarks from the Transaction Processing Performance Council (TPC) [28].

The TPC benchmarks became influential, with *DebitCredit* evolving into TPC-A and later TPC-C for OLTP [2] workloads. To address both OLTP and OLAP [3] systems, the hybrid TPC-CH benchmark was introduced [14], bridging TPC-C (OLTP) and TPC-H (OLAP). Building on TPC-H, the *Star Schema Benchmark (SSB)* was developed to measure database performance in traditional warehousing applications [23].

Over time, specialized benchmarking tools emerged to evaluate diverse database architectures. Examples include a variety of benchmarks inspired by social network graph structures including LinkBench [1] and TAOBench [8], *Jackpine* for spatial databases [25], SmartBench [17] for IoT support. LDBC is a consortium which has defined a family of benchmarks for graph data, eg [13]. YCSB was introduced to measure performance of cloud-based key-value platforms [9], and because it offers an extensible framework, later work used YCSB as the basis to incorporate other aspects including transactions [10], geospatial data [22]. Recently, [3] focuses on the

[2] OLTP - Online Transaction Processing.
[3] OLAP - Online Analytical Processing.

danger of systems becoming tuned for widely-used benchmarks, and advocates for introducing new "surprise" aspects each year.

Value Size in Benchmarking. Large object sizes have been considered in database benchmarking due to their direct impact on overall database size. Biliris [4] analyzed the performance of large object management across three database systems from 1992—EXODUS [7], Starburst, and EOS—focusing on length-changing updates. The study proposed strategies for segment size adjustments to enhance operational efficiency while noting the trade-off between performance gains and reduced storage utilization with large fixed-size segments.

In past decades, the main relational database benchmarks have predominantly employed fixed (or only slightly varying) item sizes in their setups [2]. While scaling the total database volume has been a traditional benchmarking consideration (eg "scale factor" in the TPC benchmarks), the dynamic nature of individual value sizes and their distribution over time has received less attention.

Cao et al. demonstrated in RocksDB-based production systems such as UDB, ZippyDB, and UP2X at Facebook that the distribution of key and value sizes is closely tied to specific use cases and applications [6]. A recent study considered space-optimising storage structures in Neo4j under real application loads, and found little impact on overall performance, ascribed perhaps to the limited importance of indexing in that platform [27].

4 BENCHMARK DESCRIPTION AND DESIGN

The main contribution of this paper is a new benchmark, which we call YCSB-IVS for *YCSB - Increasing Value Sizes*, which can assess the impact on database performance as the size of data items grows over time. We work within the framework of Yahoo! Cloud Serving Benchmark (YCSB[4]) [9]. This is a widely used framework to evaluate the performance of NoSQL and SQL databases and other cloud-based data delivery systems. Designed to facilitate comparative analysis, YCSB provides a set of core workloads that simulate varied real-world database uses, including read-heavy, write-heavy, and balanced scenarios. The workloads are mixes of READ, UPDATE, INSERT, DELETE and SCAN operations, with configuration parameters that control the distribution of the keys involved. The framework enables us to measure crucial performance metrics such as throughput, latency (average and tail), and scalability under different configurations. YCSB's extensibility allows for customisation of workloads, database bindings, and distributions, making it a versatile tool for assessing various aspects of data serving systems under controlled conditions. As YCSB is a benchmark that assumes only a key-value logical data model, we implement a *value-length extension* functionality, which increases the length of the value associated with a key; due to the generality of this logical data model, we can apply the benchmark to platforms with richer data models, including relations or JSON documents.

Extend Operation. To see the impact of growing value sizes on database performance, we implemented an extra operation, which will increase the size of value for a key chosen according to a specific distribution; this is similar to the original UPDATE of YCSB, in that a key is chosen from a distribution, and the associated value is replaced by a new one; unlike the provided UPDATE, our EXTEND associates to this key a new value which is longer than the previous one by the amount specified in the workload configuration as the `extendfieldlength` parameter (we have a default increase of 100 bytes).

4.1 Benchmark Methodology

The original YCSB framework begins with a *load* phase where data is generated and loaded to the data store being evaluated, and then performance is measured during a *run* phase, where simulated clients submit operations (such as READ, UPDATE, etc.) against the data store. In the original framework default configuration, the data is a collection of items all exactly 1000 bytes long. In our YCSB-IVS, we have a more complex experimental approach to capture performance when data items can grow; the initial data load is followed by a succession of *epochs*; each epoch has two phases, first an *extend* phase in which the EXTEND operation is performed many times, thus growing the length of some of the records, and then the epoch has a *run* phase where we measure throughput and latency for a mix of operations, against the data that resulted from the extend phase of this epoch. Then the next epoch begins, with further extending record lengths and then another measured run phase. In fact, we use the YCSB framework's run capability for both our extend and run phases in each epoch: our extend phase is simply a YCSB run of a workload containing 100% EXTEND operations, and our run phase has a workload with 100% READ operations. In this way, the measurements we capture in our run phase of a given epoch are all against a data set with a particular distribution of item lengths, which grows from epoch to epoch due to the extend phases. Our experiment results display the way the metrics (throughput and average latency) change as the epochs progress.

There are several pragmatic issues that arise from restrictions in YCSB or the platforms we work with, and that lead to some detailed impacts in the way we set up our benchmark.

Maximum record size. Many database systems impose restrictions on the maximum size of individual records, stemming from both design considerations and practical limitations. For example, MongoDB restricts documents to a maximum size of 16MB, as detailed in their official documentation[5]. MariaDB, while more flexible, adheres to its own constraints: the LONGTEXT data type must be used for fields exceeding 64KB[6]. However, with MariaDB's default settings, a client application can create a row larger than 16MB, yet that row cannot be successfully read due to an underlying limit on the packet size.

Given these constraints, we have designed our workload to incorporate a cap on record sizes by limiting the size of individual fields. When an "extend" operation encounters a record that has reached its maximum allowable size, it is bypassed entirely, and no update is applied.

In our workload, the default field size limit is set at 1.6MB, which, in consequence, sets a record's upper boundary on length at just below the 16MB threshold to circumvent these platform-specific limitations. This choice affords us several advantages: it provides a

[4]YCSB: https://github.com/brianfrankcooper/YCSB.git

[5]https://www.mongodb.com/docs/manual/reference/limits/#bson-documents

[6]https://mariadb.com/kb/en/text/

workable solution that sidesteps platform constraints while having minimal impact on our workload's execution.

Our heavyweight workload (described in subsection 4.2) with the Zipfian request distribution for extend operations targets a small set of records to be extended most often as shown in Figure 1, yet we found that no more than two records reached this maximum limit. This validates our design decision, demonstrating that even under stringent conditions, the limit is sufficient to prevent record expansion beyond what the database systems can handle while minimizing the impact of the limit on ordinary operation of the workload. This limitation serves as a protective measure, ensuring the robustness and reliability of our workload against the backdrop of real-world database constraints.

Extend Phase. This phase involves submitting EXTEND operations against the database, with keys chosen according to the `requestdistribution` parameter which is specified in the configuration. In our experiments, we consider both Uniform and Zipfian distributions for the keys to be extended (these distributions are in the default YCSB codebase). For the Zipfian distribution, we use the YCSB default parameter 0.99. Extending keys chosen as a Zipfian distribution means that a few data items will be extended much more frequently than others, and these popular items will therefore grow quite long as the epochs pass. An item which is chosen in an extend operation grows in length by the amount given as the `fieldlength` configuration parameter, defaulting to 100 bytes (this default is used for all the experiments we report here). We also use the data platform's backup capability to take a complete dump of the logical data state at the end of each extend phase, for use in the *clean-run* we describe later.

Run Phase. The *run* phase of an epoch in YCSB-IVS has 100% READ operations (no SCAN, INSERT, UPDATE or DELETE operations). The number of READs in an epoch is controlled by the `operationcount` configuration parameter and is chosen to be large enough so that any startup effects from running the YCSB client are amortized. In our testing, 10,000 operations were insufficient and startup effects were observed, noting that YCSB does not support a warm-up phase before measurement begins. Given that, all our experiments use 100,000 READ operations. For this phase, keys are chosen by the uniform distribution regardless of what distribution is used for the extend phase. Using a uniform distribution means that read operations will be evenly distributed across value sizes. In each epoch's run phase, we use the YCSB facility to capture the throughput and average latency metrics.

Experiment Modes. As described above, the ***main-run*** epochs in our experimental methodology involve iteratively executing *extend* and *run* phases following the initial data loading, to show how performance changes as some data items grow in length over time, through the EXTEND operations. This is a simplified version of what we have seen in real-world scenarios. However, one might consider whether the performance is based on the history of growth in some records over time, or instead is just a consequence of having records of different size in the databse state, without any change through time. So,for better understanding, we have included in YCSB-IVS methodology, another set of epoch-based measurements which aim to disentangle these issues.

In YCSB-IVS, we measure the ***clean-run*** mode, in which, in each epoch, the logical data backup taken at the end of the extend phase of *main-run* is restored to a fresh instance of the data platform, and then the run operations are submitted to this clean copy of the data. That is, in any epoch of *clean-run*, we have a database state which is logically the same as that in the corresponding epoch of *main-run*, but without any history of change over time, so the physical structures will likely be simpler. Any difference between *main-run* and *clean-run* reveals an effect that is due specifically to change in record lengths through an execution.

The baselines use only prior YCSB functionality (that is, there are no EXTEND operations, and all records have uniform length). They are measured separately, rather than after each extend phase, to show the impact of simple impacts of the scale of data volume, separate from non-uniformity. We perform the baselines for a range of scales corresponding to the data volume seen in the dump at the end of an epoch (this can in fact be calculated from the number of extend operations performed in the epochs up till then, and the extra length added in each operation). ***spread-baseline*** measures the effect of having many items of a fixed size and ***average-baseline*** measures the effect of a fixed set of long data items. In *average-baseline*, the data volume is distributed equally among the same number of records as in *main-run*; this ensures uniform item lengths which are quite long, and longer at increasing scale. The data with this uniform length is loaded into a fresh instance of the platform, and the read-only operations are run against that. In contrast *spread-baseline* for a given data volume has all records with the same length as those in the initial load of *main-run*, before any extensions; and thus the scale in data volume is achieved by having a very large number of records in *spread-baseline* (far more than in *main-run*).

As one would expect (and will mostly be confirmed in our measurements), the performance of *spread-baseline* is mostly independent of the scale of data volume; the indexing in each platform is effective, and so the reads are not much impacted when there are many more records to choose among. In contrast, each platform shows substantial slowdown with long records, though the details vary. This is expected, since each read operation accesses and outputs the whole record. In our charts, we will show the metrics for *main-run* and *clean-run* at each epoch, and also for comparison the metrics from *average-baseline* at the appropriate data volume.

Any substantial difference between the metrics in *clean-run* for an epoch, and *average-baseline* for the data volume at the end of that epoch, indicates that the platform is affected by variations in item lengths. In contrast, any difference between *clean-run* and *main-run* can be attributed specifically to the gradual growth of item lengths over time and its impact on physical storage structures. If all three modes yield similar metrics, then the determining factor is simply the total record length, rather than variations in individual item sizes.

Each result chart shows curves for a performance metric against the epoch number.

4.2 Benchmark Settings

On each platform, we consider two scales in data quantity, which we refer to as *lightweight* and *heavyweight*. Heavyweight aims for

substantially larger data scale, whereas lightweight gives a finer-grained exploration of smaller volumes, with an order of magnitude fewer records and fewer extensions performed per epoch. Table 1 outlines the configuration differences for these two experimental scales. Note that the run phase involves the same number of READ operations, no matter which scale we work at.

Table 1: Experiment configurations for different scales.

Scale	record count	extend count	read count	initial field length
lightweight	1000	10000	100000	100
heavyweight	10000	100000	100000	100

Since records start at 1000 bytes and each extend operation adds 100 bytes to a record, the lightweight workload starts with approximately 1MB of data and after 100 epochs, grows to 100MB. The heavyweight workload starts with 10MB of data and grows to 1GB. While this volume of data isn't enough to stress the caching or I/O of each system, our focus is on the cost to access records of varied lengths in cached data. By the time we have completed 100 epochs, the workloads have performed 10 million READ operations.

4.3 Benchmark Design Choices

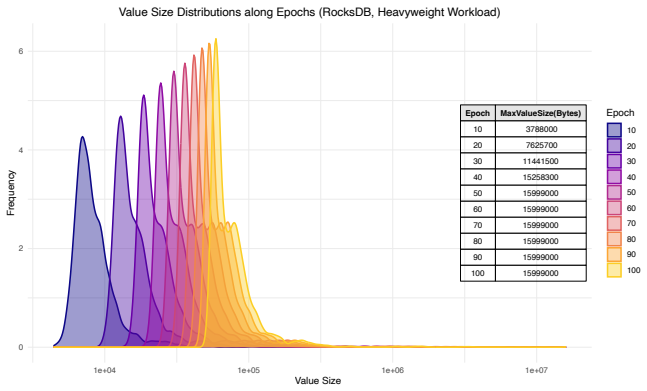


Figure 1: Value size distribution and maximum value size change along the epochs for the heavyweight workload with Zipfian distribution of extends.

Choice of Request Distributions. We employ *Uniform* and *Zipfian* distributions as our primary key distribution for the extend phases. These distributions influence the value size extensions and subsequent workload behavior as follows:

(1) **Uniform Value Extension:** Using a uniform distribution for value size extensions ensures that values across the entire key range are equally likely to be extended. This approach results in a less dispersed distribution of value sizes (though still not with all records the exactly the same length).

(2) **Skewed Value Extension with Zipfian:** In contrast, extending values based on a Zipfian distribution on the chosen keys, prioritizes the most popular keys, leading to a more skewed value size distribution. This approach simulates scenarios where a small subset of keys accumulates disproportionately larger values.

Regardless of the distribution used during the extension phase, we employ a *Uniform* distribution on keys when running the read-only workloads in the *run* phase. This choice ensures that the workload reads a representative mix of values across varying sizes, regardless of the extension strategy. For instance, even when values are extended using a Zipfian distribution, a Uniform request distribution mitigates the tendency to disproportionately access the very large values, yielding outcomes that provide a fair comparison to those observed with uniform extensions.

While additional distributional choices could be explored, we limit our analysis to these two due to resource constraints, focusing on the fundamental impact of value size variation on database performance.

As can be seen in Figure 1, the heavyweight workload with Zipfian distribution of extend operations both grows the typical record length *and* makes record lengths increasingly skewed as the epochs advance.

Performance Evaluation Matrices. We utilize *latency* and *throughput* as the primary metrics for evaluating system performance, reflecting their widespread adoption in performance analysis [29]. These metrics are recorded during both the *extend* and *run* phases of our experiments.

For each workload, YCSB provides the overall throughput and summary statistics for the latency distribution. Latency values vary across a range due to factors such as database processing time, network traffic, and other system-level overheads. To identify a representative latency measure, we analyzed various summary statistics and observed that the relative ordering of latency values remains consistent across experimental modes, irrespective of the statistic selected.

The latency values for experiments conducted on MongoDB are shown in Figure 2, highlighting this consistency across modes. Without loss of generality, we selected ***average latency*** and ***throughput*** as the primary performance metrics for our study.

5 CASE STUDIES OF THE BENCHMARK

In this section, we show the utility of our proposed benchmark, by running it on several varied platforms, reporting the results, and showing that it reveals interesting facets of the performance of these platforms when data items grow in length over time.

Platforms. To evaluate the impact of value size changes across a diverse range of database architectures, we selected three widely used database platforms for our experiments. These DBMSs, summarized in Table 2, represent a cross-section of logical data models and physical storage structures.

Infrastructure. The experiments were conducted on an AWS EC2 `m7i.xlarge` instance running Ubuntu 24.04 LTS. The machine is equipped with a dual-core Intel Xeon Platinum 8488C processor

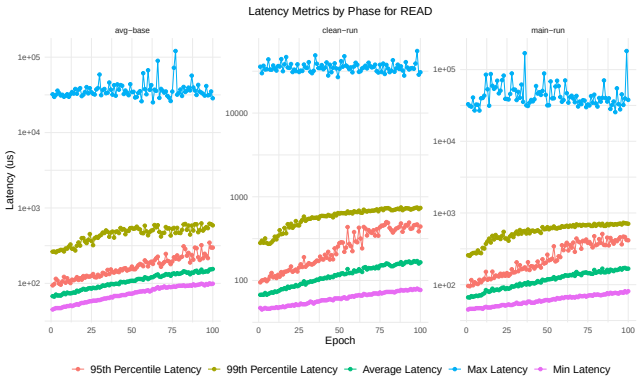


Figure 2: Behavior of latency summary statistics for MongoDB *run* phases for the heavyweight workload.

Table 2: Selected DBMSs and their characteristics.

	Database	Version	Logical Model	Storage Structure
1	MongoDB	6.0	Document	B-tree
2	MariaDB+InnoDB	10.6	Relational	B-tree
3	MariaDB+MyRocks	10.6	Relational	LSM-tree

(Sapphire Rapids architecture) with a 105 MB L3 cache and hyperthreading enabled. It includes 16 GB of RAM and a 30 GB NVMe-backed Elastic Block Store for storage. Networking is handled by an Elastic Network Adapter, and the instance is configured without a graphical interface to optimize for computational workloads.

MongoDB and MariaDB+MyRocks were both run with default settings. MariaDB+InnoDB was run with the buffer pool size adjusted by setting `innodb_buffer_pool_size` to 4GB[7].

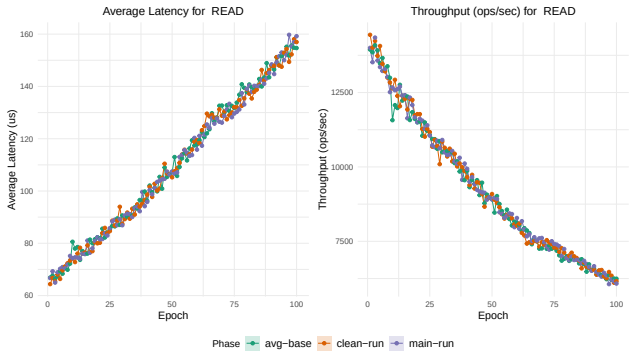
5.1 Value Size Increase in MongoDB

Baselines. In Figure 4, we present a comparative analysis of the baselines for this platform, as explained in Section 4. The first baseline, referred to as *spread-baseline*, involves obtaining the indicated total volume of data through as many records as needed, all of which have the same size (1KB) corresponding to the initial size in our benchmark runs. In contrast, *average-baseline* has the given data volume through the the same number of records as in the heavyweight benchmark (10,000), with whatever equal size if then needed for the given volume.

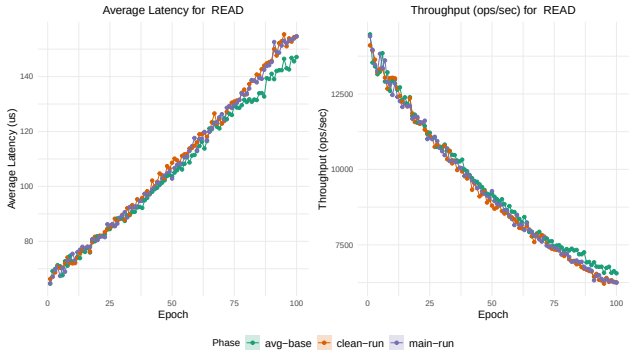
Our study examines the performance of a query workload designed to select records with a uniform key distribution across various platforms. The results show that under *spread-baseline*, MongoDB exhibits fairly stable read throughput as the data volume scales through the presence of more records. This indicates that additional records, each of consistent size, do not adversely affect MongoDB's ability to handle read operations.

In contrast, with *average-baseline* a clear near-linear degradation in latency is observed as the volume (and thus the sizes of the records) increase. This deterioration underscores the impact of record sizes on query performance. These findings demonstrate the importance of workloads that explore database performance with a variety of record sizes.

[7]https://mariadb.com/docs/server/storage-engines/innodb/operations/configure-buffer-pool/



(a) Uniform request distribution for extend operations



(b) Zipfian request distribution for extend operations

Figure 3: Query performance under growing value sizes in MongoDB with a lightweight workload.

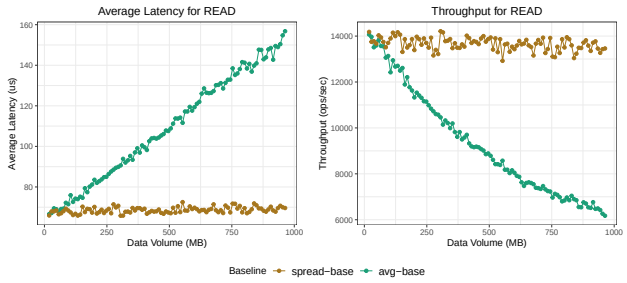
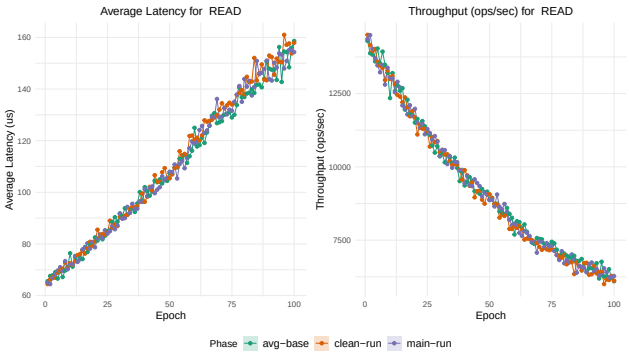
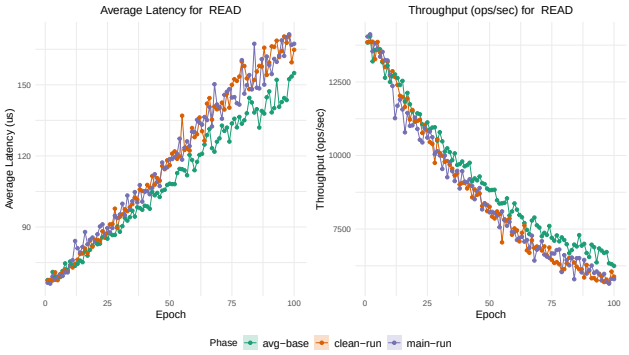


Figure 4: Query performance baselines for MongoDB, where a given data volume is created in *spread-baseline* by varying the number of fixed-size records, or in *average-baseline* by varying the size of a fixed number of records.

Heavyweight workload. We evaluate MongoDB using the heavyweight workload, as shown in Figure 5. The figure illustrates the

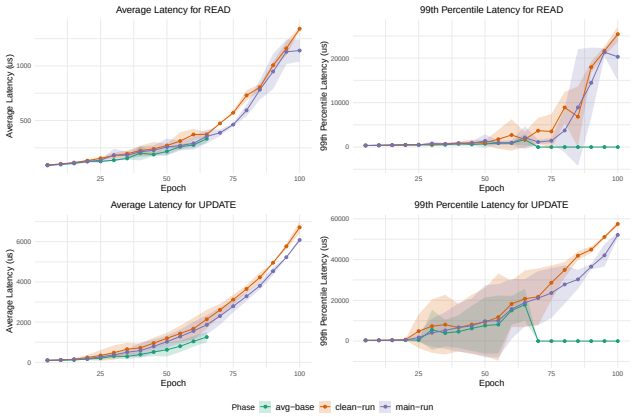


(a) Uniform request distribution for extend operations

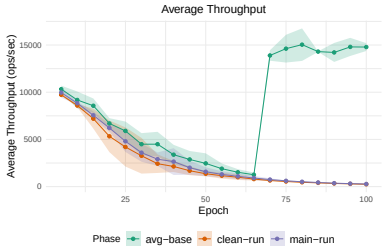


(b) Zipfian request distributions for extend operations

Figure 5: Query performance under growing value sizes in MongoDB with a heavyweight workload.



(a) Average and 99th percentile latency for READ and UPDATE operations.



(b) Average throughput.

Figure 6: Query performance of a mixed 50:50 READ/UPDATE workload under increasing value sizes in MongoDB with a heavy-weight configuration and uniform extensions.

latency trends for read-only queries over multiple epochs as the record sizes grow.

In both *main-run*, where queries are executed immediately after extending the data, and *clean-run*, where the data is dumped and reloaded into a fresh database before querying, we observe steadily increasing latency as the records grow in size. This suggests that performance degradation is primarily due to increased record sizes, rather than fragmentation of the database or the runtime state.

In Figure 5(a), where the extend operations follow a uniform request distribution, the resulting value sizes exhibit a normal distribution. We observe no significant differences in query performance between *main-run*, *clean-run*, and *average-baseline*, where all values are set to the same length while keeping the total number of queried bytes identical. This indicates that under a uniform distribution, the system handles queries over varying and constant-sized values similarly.

In contrast, Figure 5(b) presents results for a Zipfian request distribution, which causes a Zipfian distribution of value sizes. In this case, we observe a performance difference where *main-run* and *clean-run*, which have a skewed distribution of value sizes, are slower than *average-baseline*, where all values are the same size.

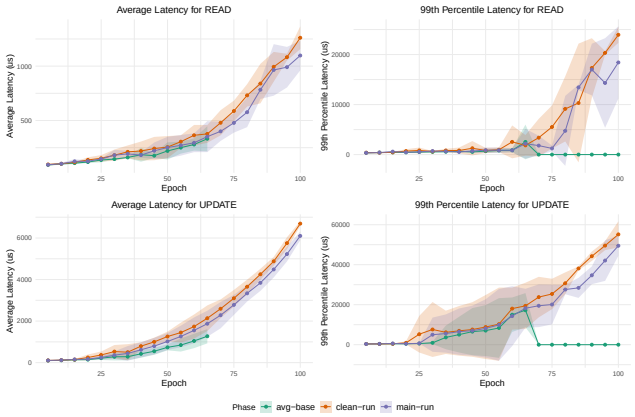
5.2 Value Size Increase in MariaDB+InnoDB

Baselines. In Figure 8, we measure the baseline performance of MariaDB+InnoDB using *spread-baseline* and *average-baseline*. The results show that under *spread-baseline*, MariaDB+InnoDB exhibits fairly stable read throughput as the number of records grows. In contrast, with *average-baseline* a clear degradation in throughput is observed as the sizes of the records increase. The overall increase appears to be slightly sublinear but with a small step up at around 400MB and a marked increase at around 600MB.

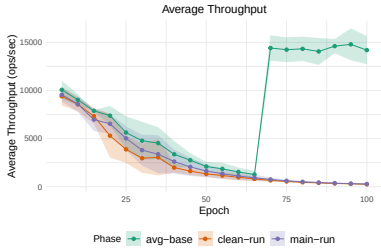
Lightweight Workload. Figure 9 illustrates the performance of MariaDB+InnoDB storage engine under lightweight workload. As observed with other systems, there is an overall increase in latency and a reduction in throughput across the epochs as the value size increases.

One notable difference in Figure 9(a) is that the increase in latency is nonlinear. After around epoch 40, performance decreases more slowly as the average record size grows.

As with MongoDB, in Figure 9(b) we observe that MariaDB+-InnoDB also shows a performance difference where *main-run* and *clean-run*, which have a skewed distribution of value sizes, are slower than *average-baseline*.



(a) Average and 99th percentile latency for READ and UPDATE operations.



(b) Average throughput.

Figure 7: Query performance of a mixed 50:50 READ/UPDATE workload under increasing value sizes in MongoDB with a heavy-weight configuration and zipfian extensions.

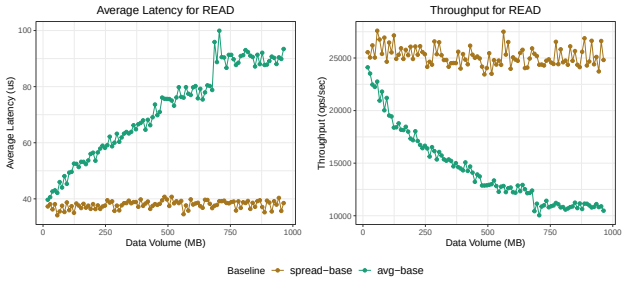
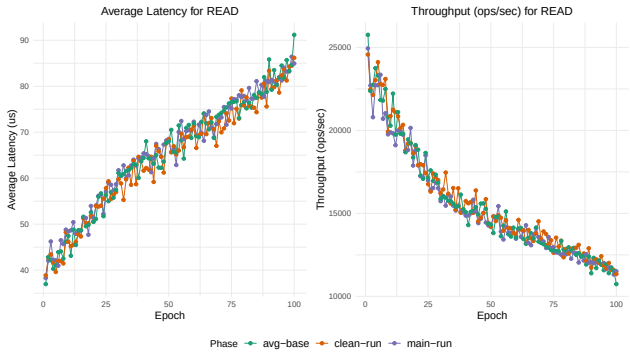
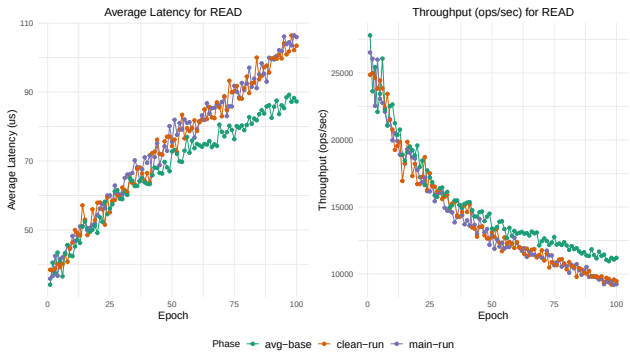


Figure 8: Query performance baselines for MariaDB+InnoDB, where a given data volume is created in *spread-baseline* by varying the number of fixed-size records, or in *average-baseline* by varying the size of a fixed number of records.

Heavyweight Workload. To investigate whether the effects observed in Figure 9 hold at larger data volumes, we scaled the data size by a factor of 10 and again evaluated MariaDB+InnoDB, as shown in Figure 10. With a uniform distribution of extend operations, as shown in Figure 10(a), large steps are seen in performance



(a) Uniform request distribution for extend operations



(b) Zipfian request distribution for extend operations

Figure 9: Query performance under growing value sizes in MariaDB+InnoDB with a lightweight workload.

for all three workloads around epoch 50 and epoch 70. Eventually, they return to the linear trend.

With the Zipfian distribution of extend operations in Figure 10(b), *main-run* and *clean-run*, which have a skewed distribution of value sizes, maintain the linear trend of slowing down as the data volume increases. In contrast, *average-baseline* slows less rapidly from about epoch 30. As a consequence, *main-run* is about 20% slower by epoch 100.

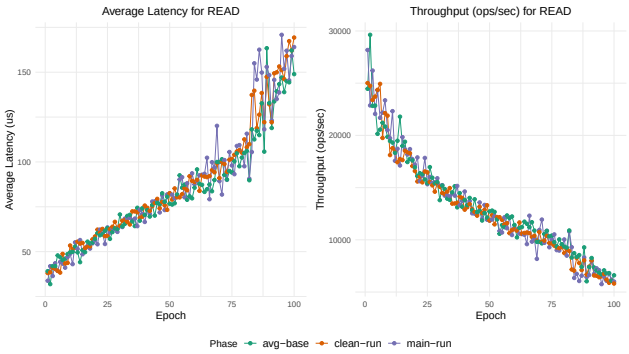
5.3 Value Size Increase in MariaDB+MyRocks

Light-weight Workload

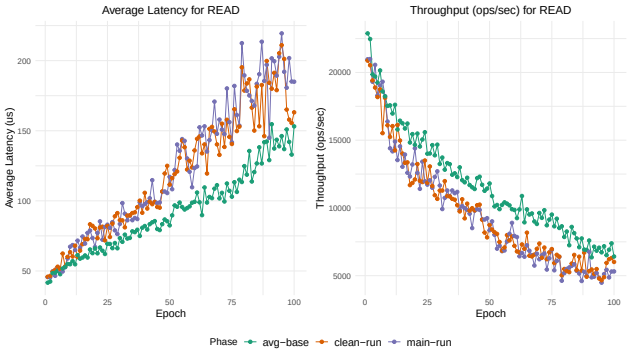
Heavyweight Workload

Baselines. In Figure 14, we measure the baseline performance of MariaDB+MyRocks using *spread-baseline* and *average-baseline*. Under *spread-baseline*, MariaDB+MyRocks exhibits slowly degrading read throughput as the number of records grows, with higher variance than the other systems we tested. With *average-baseline* there is a clear degradation in throughput is observed as the sizes of the records increase and high variance visible in the latency.

Uniform Distribution of Extend Operations. During uniform read requests, the average latency and throughput remained consistent and comparable across all three variants for slightly more than

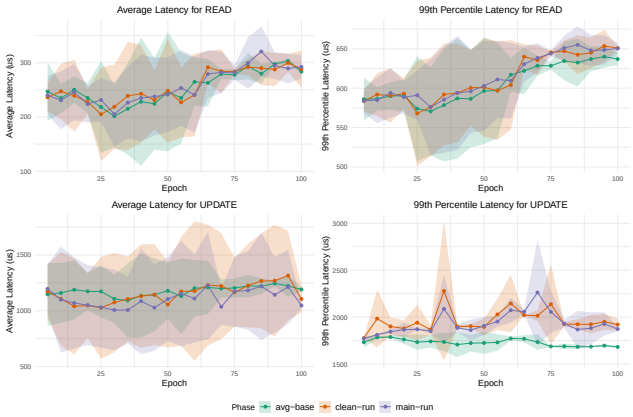


(a) Uniform request distribution for extend operations

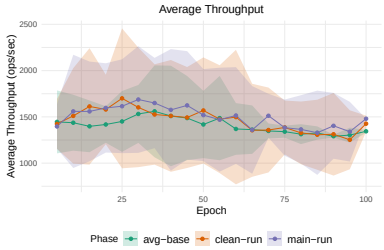


(b) Zipfian request distribution for extend operations

Figure 10: Query performance under growing value sizes in MariaDB+InnoDB with a heavyweight workload.



(a) Average and 99th percentile latency for READ and UPDATE operations.



(b) Average throughput.

Figure 11: Query performance of a mixed 50:50 READ/UPDATE workload under increasing value sizes in MariaDB+InnoDB with a heavy-weight configuration and uniform extensions.

20 extension cycles as seen in Figure 15(a). Beyond this point, the measures for records whose size varies (i.e., *main-run* and *clean-run*) exhibited a noticeable performance decline, whereas the *average-baseline* variant maintained a steady but slower degradation in read efficiency, similar to the trend observed during the initial 20 cycles.

Varying Value Sizes. For the *main-run* and *clean-run* variants, value sizes extended according to a uniform distribution. Consequently, not all values had the same size; instead, the distribution included larger and smaller values, with most sizes clustering around the mean. Larger-than-average values required disproportionately more time to read compared to smaller-than-average values, thereby degrading performance in terms of average latency and throughput, as observed in Figure 15(a) for the *main-run* and *clean-run* modes.

Overhead of querying data with history. Figure 15(b) illustrates the results of extending the value sizes in each epoch using a Zipfian distribution, where a small proportion of keys were extended more frequently compared to the uniform case. In this scenario, *clean-run* exhibited only a slight deviation from *average-baseline*. Some overhead was noted in *main-run* in Figure 15(a), but this became much clearer in Figure 10(b) when extend operations used a Zipfian distribution. Here, results deviated significantly in *main-run* after 20 epochs. Beyond this point, the average read latency in *main-run* increased by more than an order of magnitude compared to its counterpart in *clean-run*, where a fresh copy of the data was queried.

5.4 Performance comparison of extend operations

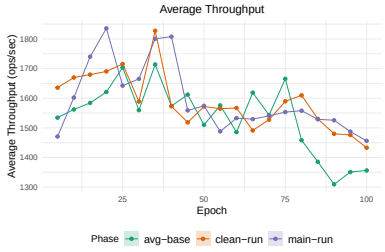
Until now, we have focused on the performance of querying data of varying sizes. For completeness, in Figure 17 we show the performance of the extend operations themselves on each of the platforms we tested.

An interesting point that can be observed in the heavyweight workload with Zipfian distribution of extend operations is that MongoDB has a step change to faster performance at epoch 46 and epoch 89. These are the points where the two largest documents reach the 16MB limit, so subsequent extend operations on those documents are skipped.

6 LESSONS LEARNED

Varying number of small records. All systems handle increasing numbers of fixed-sized records, as measured in *spread-baseline*, with minimal overhead as seen in Figures 4, 8 and 14.

(a) Average and 99th percentile latency for READ and UPDATE operations.



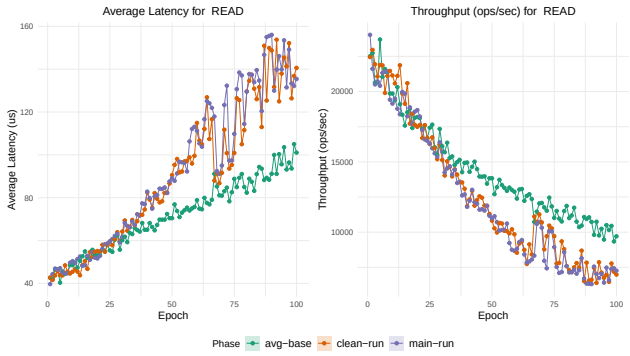
(b) Average throughput.

Figure 12: Query performance of a mixed 50:50 READ/UPDATE workload under increasing value sizes in MariaDB+InnoDB with a heavy-weight configuration and zipfian extensions.

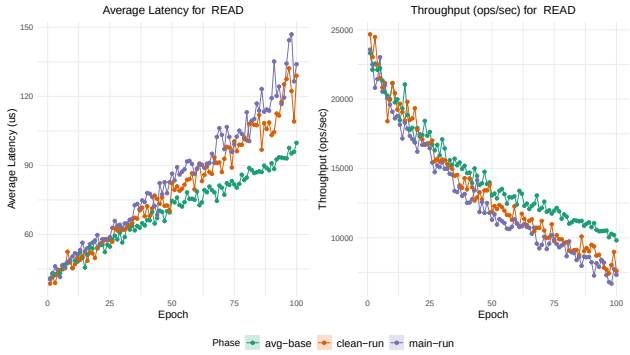
Growing record size uniformly. When record sizes are equally large, as measured in *average-baseline*, read operations slow in all systems. The response is linear in MongoDB as seen in Figure 4, slightly sublinear with some step changes in MariaDB+InnoDB as seen in Figure 8, and superlinear with high variance in MariaDB+-MyRocks as seen in Figure 14.

Varying record sizes in a narrow range. With uniformly applied extend operations, the lengths of the values have a normal distribution over a relatively narrow range, and across all systems tested we see minor differences between *average-baseline* and *clean-run* in Figure 5(a) and Figure 10(a). However, with MariaDB+MyRocks in Figure 15(a), we do see higher variance in *main-run*, discussed below.

Skewed range of record sizes. As noted in subsection 5.1, we observed that with a skewed distribution of value sizes, the performance was slightly worse than *average-baseline*, where all values are the same size. This holds even though the benchmark is designed so the total volume of data read is the same for both workloads. This behavior suggests the presence of a nonlinear cost when querying larger records, as the workload includes a small number of disproportionately large values.



(a) Uniform request distribution for extend operations



(b) Zipfian request distribution for extend operations

Figure 13: Query performance under growing value sizes in MariaDB+MyRocks with a lightweight workload.

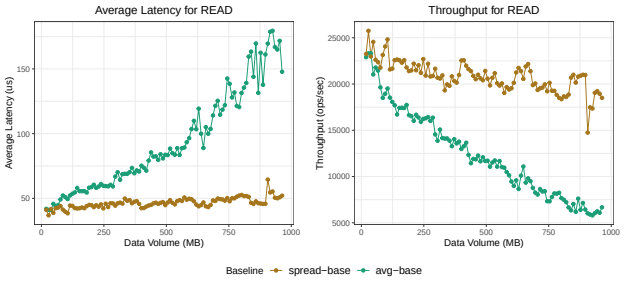
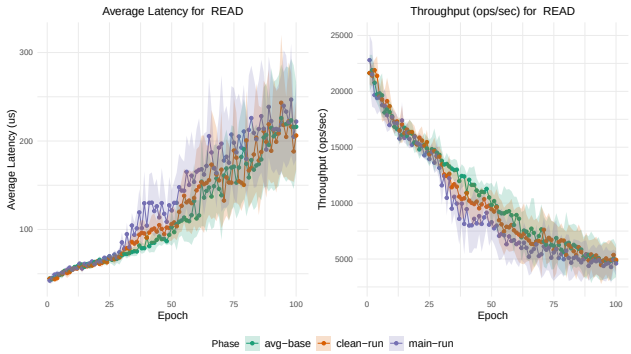
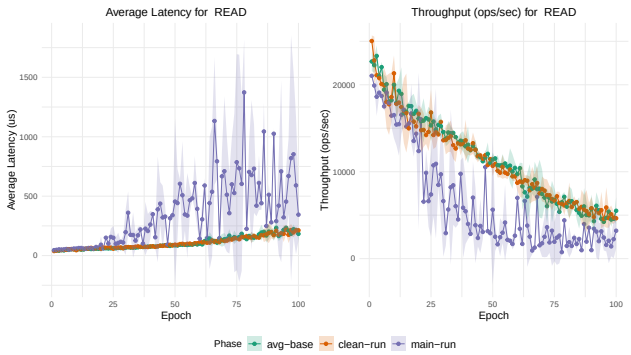


Figure 14: Query performance baselines for MariaDB+-MyRocks, where a given data volume is created in *spread-baseline* by varying the number of fixed-size records, or in *average-baseline* by varying the size of a fixed number of records.

Overall, the results demonstrate that increasing record sizes lead to steadily growing query latencies in MongoDB, with Zipfian distributions introducing additional performance overhead likely due to the nonlinear access costs of very large records.

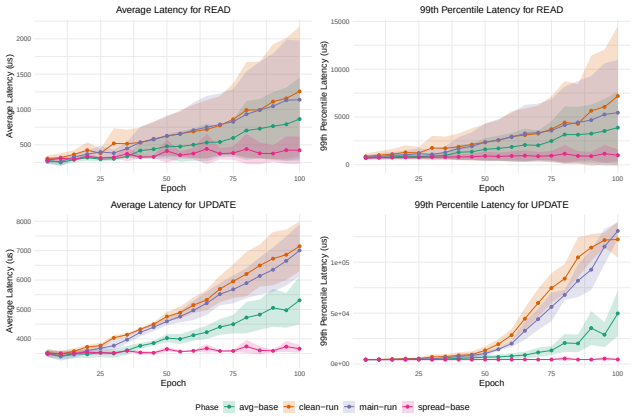


(a) Uniform request distribution for extend operations

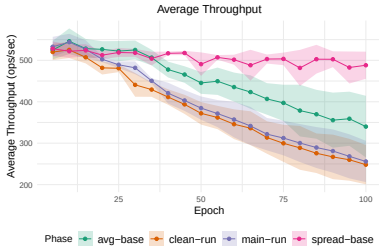


(b) Zipfian request distribution for extend operations

Figure 15: Query performance under growing value sizes in MariaDB+MyRocks with a heavyweight workload.



(a) Average and 99th percentile latency for READ and UPDATE operations over 5 trials.



(b) Average throughput over 5 trials.

Figure 16: Average query performance (over 5 trials) of a mixed 50:50 READ/UPDATE workload under increasing value sizes in MariaDB+MyRocks with a heavy-weight configuration and uniform extensions.

Step changes in performance. With a uniform distribution of extend operations in MariaDB+InnoDB, as seen in Figure 10(a), large steps are seen in performance for all three workloads around epoch 50 and epoch 70. Eventually, they return to the linear trend. We don't have an explanation for this phenomenon but it was common to all workloads.

Uniform Request Distribution and Compaction Effects. The steady performance observed during the initial 20 extension cycles in the uniform request distribution in Figure 15(a) can likely be attributed to RocksDB's handling of its LSM tree structure. As values grow, read costs naturally increase due to larger I/O operations, but RocksDB's compaction mechanism helps mitigate the immediate impact. However, once the extension cycles exceed a certain threshold, the LSM tree begins to accumulate more data at its higher levels, causing greater read amplification. This is evident in the noticeable performance decline for the variants *main-run* and *clean-run* beyond the 20-cycle mark. In contrast, *average-baseline*, with its uniform value sizes, avoids this issue, leading to a more gradual decline in performance.

Impact of Non-Uniform Value Sizes. The performance degradation observed in *main-run* and *clean-run* compared to the baseline of *average-baseline* can be explained by the non-uniform distribution of value sizes. Larger-than-average values not only increase the cost of individual reads but can also disrupt RocksDB's compaction behavior. Compaction events, which are designed to optimize the LSM-tree by merging and reorganizing data across levels, may struggle with significantly larger values. These larger records can increase write amplification during compaction, which indirectly affects read performance due to system resource contention.

Dramatic slowdown of queries against historical data. The improved efficiency observed in the *clean-run* mode, compared to the *main-run* mode, highlights the importance of the physical state of the database in query performance. With LSM trees, a fresh database, loaded with a single version of each record, benefits from a simpler LSM tree structure with no fragmented or unbalanced levels. This reduces read- and write-amplification, resulting in improved query efficiency. These results suggest that periodic reorganization or targeted compactions could be a practical optimization strategy for workloads that involve dynamically growing values.

The observed performance decline under higher data and operation volumes could be attributed to several factors, such as increased resource contention (CPU, memory, I/O) or changes in caching behavior. The significant increase in latency observed in the Zipfian extension scenario in Figure 15(b) after 20 rounds highlights the

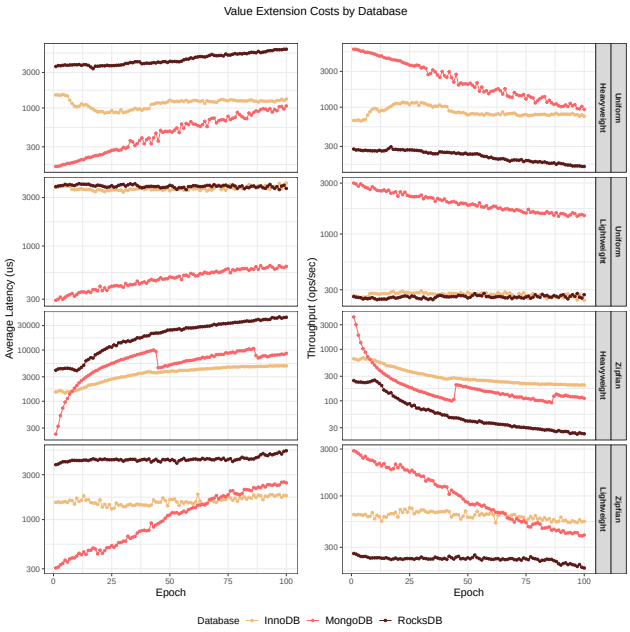


Figure 17: Extend operation performance across platforms.

impact of data skew on performance. This is likely due to the fact that the "hot" keys have many copies of increasingly large values, which can lead to inefficiencies in data retrieval and caching.

These learnings underscore the importance of considering value size distributions in database performance evaluation, including workloads involving data growth and skew. Future research could explore the impact of different value size distributions on a range of update workloads in addition to queries, the effectiveness of various optimization techniques, and the development of benchmarks that better reflect real-world scenarios with evolving data characteristics.

7 THREATS TO VALIDITY

As we introduce a novel approach to benchmarking databases with dynamic value sizes, certain threats to validity must be acknowledged.

External Validity. In real-world scenarios, value size changes arise from complex interactions between users and systems at both client and server ends. A potential threat to validity is that the temporal value changes simulated in our experiments may not fully represent such behaviors, which could affect the generalizability of our findings for comparing databases or informing decision-making.

To address this, we employed *Uniform* and *Zipfian* request distributions to model common access patterns, ensuring that both frequently accessed and randomly accessed values were reasonably represented. Furthermore, database performance is influenced by numerous factors, such as transaction processing overheads [31], making benchmarking results sensitive to environmental conditions.

To mitigate these risks, we isolated the experimental environment by using a dedicated server for benchmarking, conducted repeated sequential runs, and ensured that factors such as memory utilization and network bandwidth were controlled or their effects were randomized. This approach minimizes external interference and enhances the reliability of our results.

Internal Validity. A potential threat to internal validity is that the benchmarking results may not be applicable for all database systems that exist. To mitigate selection bias, we carefully chose three widely used databases – MongoDB, MariaDB+InnoDB, and MariaDB+MyRocks– that represent a diverse range of database architectures. This selection encompasses both B-tree and LSM-tree structures, relational and key-value store paradigms, and SQL-based as well as other query interfaces, ensuring a representative cross-section of modern database systems.

Another concern is the impact of resource allocation on efficiency measures. Varying resource configurations across systems could skew performance results. To address this, we standardized the system configurations for all benchmarks and used default database settings to ensure consistency. This approach isolates the effects of value size changes, aligning the results with the focus of our study.

Construct Validity. Accurate measurement is critical to our study, as it focuses on evaluating performance. To ensure validity, we selected *average latency* and *throughput*, widely recognized metrics in database benchmarking, as evidenced by their adoption in benchmarks like TPC and YCSB. This combination mitigates mono-method bias by providing complementary perspectives on performance, helping to identify potential impacts of confounding factors.

Additionally, to ensure unbiased measurements, we isolated the server during benchmarking and avoided any manual intervention after the benchmarking processes were initiated. This setup ensures that the results accurately reflect the system's behavior without external interference.

8 CONCLUSION

This is the first paper to introduce a unique workload involving dynamic growth in sizes of values, highlighting the importance of its consideration for building practical storage systems. We have shown the impacts of our workload on document stores and relational databases, with either tree or LSM-based storage structures. using throughput and average latency as the performance measures. In general, we observe that the growth in value sizes of a specific subset of keys poses challenges that the existing benchmarks fail to uncover. Given the presence of such a pattern of growth in practical applications, using our workload would not only help in making right choice for the database but also in implementation of databases to cater to such use cases.

REFERENCES

[1] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[2] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. 2020. Database benchmarking for supporting real-time interactive querying of

large data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1571–1587.

[3] Lawrence Benson, Carsten Binnig, Jan-Micha Bodensohn, Federico Lorenzi, Jigao Luo, Danica Porobic, Tilmann Rabl, Anupam Sanghi, Russell Sears, Pinar Tözün, and Tobias Ziegler. 2024. Surprise Benchmarking: The Why, What, and How. In *Proceedings of the Tenth International Workshop on Testing Database Systems, DBTest 2024, Santiago, Chile, 9 June 2024*. ACM, 1–8. https://doi.org/10.1145/3662165.3662763

[4] Alexandros Biliris. 1992. The performance of three database storage structures for managing large objects. *ACM SIGMOD Record* 21, 2 (1992), 276–285.

[5] Dina Bitton, David J DeWitt, and Carolyn Turbyfill. 1983. *Benchmarking database systems-A systematic approach*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[7] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. 1986. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 91–100.

[8] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, et al. 2022. Taobench: An end-to-end benchmark for social network workloads. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1965–1977.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*. IEEE Computer Society, 223–230. https://doi.org/10.1109/ICDEW.2014.6818330

[11] Dgraph Labs. 2025. Dgraph: Distributed Graph Database. https://dgraph.io/ Accessed: 2025-01-30.

[12] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. 1992. Query optimization in a heterogeneous dbms. In *VLDB*, Vol. 92. 277–291.

[13] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 619–630. https://doi.org/10.1145/2723372.2742786

[14] Florian Funke, Alfons Kemper, and Thomas Neumann. 2011. Benchmarking hybrid oltp&olap database systems. (2011).

[15] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. 1996. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System.. In *VLDB*, Vol. 96. Citeseer, 3–6.

[16] Jim Gray. 1993. Database and transaction processing performance handbook.

[17] Peeyush Gupta, Michael J Carey, Sharad Mehrotra, and oberto Yus. 2020. Smartbench: A benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1807–1820.

[18] Ali R Hurson, Simin H. Pakzad, and J-B Cheng. 1993. Object-oriented database management systems: evolution and performance issues. *Computer* 26, 2 (1993), 48–58.

[19] Stratos Idreos and Mark Callaghan. 2020. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2667–2672.

[20] Balakrishna R Iyer and David Wilhite. 1994. Data compression support in databases. In *VLDB*, Vol. 94. 695–704.

[21] Manish Jain. 2021. Dgraph: Synchronously Replicated„ Transactional and Distributed Graph Database. https://github.com/hypermodeinc/dgraph/blob/master/paper/dgraph.pdf

[22] Suneuy Kim, Yvonne Hoang, Tsz Ting Yu, and Yuvraj Singh Kanwar. 2023. GeoYCSB: A Benchmark Framework for the Performance and Scalability Evaluation of Geospatial NoSQL Databases. *Big Data Res.* 31 (2023), 100368. https://doi.org/10.1016/J.BDR.2023.100368

[23] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer, 237–252.

[24] Nikos Pelekis, Babis Theodoulidis, Ioannis Kopanakis, and Yannis Theodoridis. 2004. Literature review of spatio-temporal database models. *The Knowledge Engineering Review* 19, 3 (2004), 235–274.

[25] Suprio Ray, Bogdan Simion, and Angela Demke Brown. 2011. Jackpine: A benchmark to evaluate spatial database performance. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1139–1150.

[26] Michael Shapiro and Ethan Miller. 1999. Managing databases with binary large objects. In *16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (Cat. No. 99CB37098)*. IEEE, 185–193.

[27] Georgios Theodorakis, James Clarkson, and Jim Webber. 2024. An Empirical Evaluation of Variable-length Record B+Trees on a Modern Graph Database System. In *40th International Conference on Data Engineering, ICDE 2024 - Workshops, Utrecht, Netherlands, May 13-16, 2024*. IEEE, 343–349. https://doi.org/10.1109/ICDEW61823.2024.00051

[28] Transaction Processing Performance Council (TPC). 2024. TPC Homepage. https://www.tpc.org/. Accessed: 2024-12-20.

[29] Xin Wang, Henning Schulzrinne, Dilip Kandlur, and Dinesh Verma. 2008. Measurement and analysis of LDAP performance. *IEEE/ACM Transactions On Networking* 16, 1 (2008), 232–243.

[30] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The implementation and performance of compressed databases. *ACM Sigmod Record* 29, 3 (2000), 55–67.

[31] S. Bing Yao, Alan R. Hevner, and Helene Young-Myers. 1987. Analysis of database system architectures using benchmarks. *IEEE transactions on software engineering* 6 (1987), 709–725.