KVBench: A Key-Value Benchmarking Suite

Zichen Zhu
zczhu@bu.edu
Boston University

Arpita Saha
arpitasaha@brandeis.edu
Brandeis University

Manos Athanassoulis
mathan@bu.edu
Boston University

Subhadeep Sarkar
subhadeep@brandeis.edu
Brandeis University

ABSTRACT

Key-value stores are at the core of several modern NoSQL-based data systems, and thus, a comprehensive benchmark tool is of paramount importance in evaluating their performance under different workloads. Prior research reveals that real-world workloads have a diverse range of characteristics, such as the fraction of point queries that target non-existing keys, point and range deletes, as well as, different distributions for queries and updates, all of which have very different performance implications. State-of-the-art key-value workload generators, such as YCSB and db_bench, fail to generate workloads that emulate these practical workloads, limiting the dimensions on which we can benchmark the systems' performance.

In this paper, we present KVBench, a novel synthetic workload generator that fills the gap between classical key-value workload generators and more complex real-life workloads. KVBench supports a wide range of operations, including point queries, range queries, inserts, updates, deletes, range deletes, and among these options, inserts, queries, and updates can be customized by different distributions. Compared to state-of-the-art key-value workload generators, KVBench offers a richer array of knobs, including the proportion of empty point queries, customized distributions for updates and queries, and range deletes with specific selectivity, constituting a significantly flexible framework that can better emulate real-world workloads.

ACM Reference Format:
Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest '24), June 9, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3662165.3662765

1 INTRODUCTION

Key-Value Stores are Everywhere. Key-value-based NoSQL data systems are widely used across the industry, such as RocksDB [14], LevelDB [15], FASTER [7], MongoDB [21], SplinterDB [9], Cassandra [2], HBase [3], Redis [23], Oracle NoSQL [22], Scalaris [32], and OrientDB [25]. Even in relational databases, key-value stores can still be used as the underlying storage engines (e.g., RocksDB in TiDB [16] and Pebble in CockroachDB [8]) or as key-value indexes to improve database performance. However, with the rapid growth of new key-value stores, it becomes harder for practitioners to pick the most appropriate one for their applications. To ensure an apples-to-apples comparison across existing key-value stores, a comprehensive benchmark tool is essential to investigate the performance implications of different workloads.

Table 1: Comparison of KVBench with other benchmarks

		benchmark		
		YCSB	db_bench	KVBench
operation	insert	✓	✓	✓
	update	✓	✓	✓
	non-empty point query	✓	✓	✓
	empty point query	×	✓	✓
	range query	✓	✓	✓
	non-empty point delete	×	✓	✓
	empty point delete	×	✓	✓
	range delete	×	✓	✓
distribution	uniform	✓	×	✓
	normal	×	✓	✓
	beta	×	×	✓
	Zipfian	✓	×	✓

State-of-the-Art Workload Generators Support Limited Workload Characteristics. The most commonly used benchmark tool for NoSQL systems is YCSB [10], which can generate workloads with inserts, updates, point queries, and range queries with a specified distribution. However, YCSB does not support the whole range of basic operations in key-value stores such as empty point queries and deletes. While db_bench [13] in RocksDB, another existing workload generator, supports the above basic operations, it does not allow deletes to be mixed with query/update workloads in an interleaved manner. Besides, db_bench is designed and implemented specifically for testing RocksDB performance, which is also not applicable to other key-value stores. Below, we discuss three main challenges state-of-the-art workload generators face when emulating real-life workloads.

Challenge 1: Empty point queries are not well supported. Point queries can be classified into *empty* point queries that target keys that do not exist in or are deleted from the database and *non-empty* point queries that target keys with at least one occurrence in the database [27, 31]. A comprehensive workload generator should distinguish these two operations because they have different performance implications [12, 18]. Non-empty point queries must access the target data to obtain the associated value, while empty point queries can terminate much earlier due to metadata (e.g., Bloom Filters and Zonemaps) [26, 28]. For example, consider a scenario with two e-commerce platforms, where one (platform A) has a popular commodity, while the other (platform B) does not. Without knowing which platform has the popular commodity in advance, the workload in platform B may contain a significant number of empty point queries.

DBTest '24, June 9, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0669-1/24/06
https://doi.org/10.1145/3662165.3662765

Challenge 2: Inserts, Updates, and Point Queries may have Different Distributions. Existing workload generators produce inserts, updates, and queries on keys that follow the same distribution, which does not necessarily reflect real-life workloads. Further, as empty point queries are not well-supported, prior approaches cannot generate different distributions for empty vs. non-empty point queries. Consider a social media application where users post and search certain popular keywords or hashtags but may not frequently search for non-existing hashtags. In this case, updates and non-empty point queries should follow a skewed distribution (e.g., Zipfian), while empty point queries follow a uniform distribution.

Challenge 3: Deletes may be Interleaved with Updates and Queries. Invalidating an entry is a common operation in key-value stores. In most cases, a delete is issued to an existing key in the database [29, 30, 35]. For example, in a social media platform, due to privacy or other regulation issues, posts can be frequently deleted, which could happen together with other new posts or searches. In fact, 6% of all operations processed by ZippyDB, a RocksDB use case at Meta, are deletes [6]. In addition, each accessed KV-pair gets 1 delete operation on average. However, state-of-the-art workload generators that support deletes, either do not support deletes on existing keys (YCSB [10]) or do not generate deletes interleaved with point queries and updates (db_bench [13]).

KVBench: A Distribution-Aware Key-Value Benchmark. In this paper, we present **KVBench**, a scalable and distribution-aware workload generator implemented in C++. KVBench can generate a diverse spectrum of key-value workloads that are common in real life but not supported by state-of-the-art benchmarks. Similarly to the existing benchmarks, KVBench supports variable key size, variable value size, and variable fractions of operations like inserts, updates, point queries, and range queries. In addition, KVBench can distinguish between empty point queries and non-empty point queries, and it exposes the proportion of empty queries as a knob. KVBench can also generate workloads with point and range deletes interleaved with other operations, such as inserts, updates, and queries. We design KVBench to *support different key distributions per operation type*. For example, in the same workload, updates may follow a Zipfian distribution, whereas the point queries may be generated uniformly. KVBench supports uniform, normal, beta, and Zipfian distributions with configurable parameters. Together, these features make KVBench a flexible workload generator with a high degree of freedom, which, in turn, enables the users to generate vastly diverse key-value workloads that closely relate to real-world use cases. Table 1 summarizes KVBench's new features for benchmarking key-value stores compared to YCSB [10] and db_bench [13].

Contributions. In summary, our contributions are as follows.

(1) We implement two query generators that respectively support empty and non-empty point queries with the same key size. Empty point queries can be mixed with non-empty ones within the same workload. Other ingestion, deletion, and scan operations may also be interleaved manner with the point queries.
(2) We separate the generation of empty and non-empty point queries and updates into three independent generators. This way, the keys queried and the keys updated can follow a different distribution. The distribution of the operations (uniform, normal, beta, Zipfian) is specified by the users.
(3) We keep track of the existing entries when generating the workload, which allows us to interleave deletes with other operations, ensuring that all keys to be deleted exist in the database.
(4) We put everything together to build KVBench[1]. We present a methodology for generating workloads that are not supported by existing benchmarks like YCSB and db_bench, and we present a new set of benchmark workloads inspired by real-life applications. Although KVBench's goal is not to replace existing benchmarks, it can mimic several of their workloads while offering more knobs to generate new workloads.

2 RELATED WORK

In this section, we review prior work on workload generators and benchmarking tools for NoSQL systems.

YCSB. The Yahoo! Cloud Serving Benchmark (YCSB) suite [10] is a popular framework for benchmarking cloud-native NoSQL systems. It consists of a workload generator and a database interface which interact with each other via parallel client threads. These parts can be decoupled and extended to be tested on any workload generator and any database layer using the YCSB API. YCSB implements three types of workload: `CoreWorkload`, `RestWorkload`, `TimeSeriesWorkload`, among which `CoreWorkload` is designed for database benchmarking, `RestWorkload` is for `RESTFUL` operations in web services, and `TimeSeriesWorkload` is a specialized workload for time series data. There also exists a `C++` version YCSB, YCSB-C [17], that only implements `CoreWorkload` module in YCSB. We focus on the `CoreWorkload` for database benchmarking in this paper. In `CoreWorkload`, YCSB can create workloads with point queries, inserts, updates, and scans where point queries and updates can follow a specified distribution (including uniform, exponential, sequential, Zipfian, latest, and hotspot). However, `CoreWorkload` does not support delete and empty point queries, and it forces updates and queries to follow the same distribution.

RocksDB's db_bench. The db_bench benchmarking tool is integrated with the RocksDB codebase and comes as an add-on for testing RocksDB performance [13]. It supports insert, update, point query, range scan, point delete, and range delete operations. It also supports a mixed workload between point queries and updates with customized distribution (e.g., power distribution). Unlike YCSB, db_bench supports empty point queries and range deletes, and further allows the users to specify the number of keys to be deleted in range deletes. However, the implementation for empty point queries is done by increasing the key size for point queries by 1. This prevents db_bench from generating an interleaved workload with both empty and non-empty point queries. db_bench also forces the distribution to be the same for both updates and point queries, which limits the scope of real-world workloads that db_bench can emulate. Further, while db_bench supports point and range deletes, these operations cannot be mixed with other operations, such as point queries and updates, in an interleaved workload.

ForestDB-Benchmark. The ForestDB-benchmark [11] is developed for comparing ForestDB [1] with other key-value stores, such as LevelDB [15], RocksDB [14], and WiredTiger [34]. The associated API wrapper supports classical key-value operations, such as

[1]The codebase of KVBench is available at https://github.com/BU-DiSC/kv-bench.

inserts, point queries, and range queries, as well as, specific distribution (uniform, normal, and Zipfian) for the point operations. For each point operation (read or write), the associated key is randomly generated. A unique feature of the ForestDB-benchmark is that it groups the same type of operation (reads or writes) into a batch (where the batch size can follow a specified distribution) and tries to reduce duplicate operations within a batch by keeping track of all the operations within a batch. In other words, queries on the same key within a batch can only occur with low probability and inserts are almost always on unique keys. However, the ForestDB-benchmark does not explicitly support updates and empty point queries. It also has the same limitations as YCSB, i.e., it forces the reads and writes to follow the same distribution and does not support point or range deletes.

TPC and Other Benchmarks for Relational Systems. TPC [5, 33] is another popular benchmark for relational databases. It contains predefined schemas for a set of tables and a set of built-in queries with complex join operations. There are also some other similar benchmarks or datasets (e.g., LinkBench [4] and JOB [19]) that are widely used to benchmark relational databases. While we can use these existing benchmarks for relational databases that use NoSQL systems (such as MyRocks [20]) as the underlying storage engines, they do not accurately reflect the performance of NoSQL systems due to additional query processing overhead, particularly, when transforming classical SQL queries into key-value operations during the execution of SQL queries. For the same reason, benchmarking tools designed for dataframes like FuzzyData [24] cannot be used here either to benchmark NoSQL systems.

3 THE KVBENCH BENCHMARK SUITE

KVBench is implemented in C++, and it is a scalable and distribution-aware key-value workload generator that supports a diverse array of new workload characteristics that are not supported by the state-of-the-art. First, it differentiates empty and non-empty PQs and allows the users to specify the fraction of empty queries when generating workloads. Second, it allows different operations (e.g., inserts, updates, and queries) in the same workload to follow different distributions. Further, it can generate workloads with point and range deletes interleaved with other key-value operations. Users can simply pass arguments to the command line as input to KVBench, and KVBench will output a workload file with the input specification. In this section, we introduce the functionality of KVBench in detail and discuss some of its advanced features.

3.1 Basic Key-Value Operations

We now discuss all the basic key-value operations and their implementation in KVBench. All basic operations can be specified via the command line, as summarized in Table 2.

Insert. This operation is used to write a *new* key-value pair to the database. During the workload generation, we maintain a hash set (HS) to track all existing keys in the database and ensure that every key to be inserted does not exist in the current database. Specifically, whenever we randomly generate a key k to be inserted, we check if $k \in HS$. If this check returns true, we ignore k and generate a new key. If not, then we insert the key into the hash set HS, generate the *value* for k, and write the key-value pair to the output. The number of inserts is specified by `-I` in the command line.

Table 2: Basic key-value operations in KVBench.

args	description	min, max; default
-I	number of inserts	[0, UINT64_MAX]; 0
-U	number of updates	[0, UINT64_MAX]; 0
-D	number of point deletes	[0, UINT64_MAX]; 0
-R	number of range deletes	[0, UINT64_MAX]; 0
-y	range delete selectivity	(0.0, 1.0]; 0
-Q	number of point queries	[0, UINT64_MAX]; 0
-S	number of range queries	[0, UINT64_MAX]; 0
-Y	range query selectivity	(0.0, 1.0]; 0
-Z	fraction of empty point queries	[0.0, 1.0]; 0

Update. This operation is used to overwrite the *value* of an existing key-value pair in the database, which means that the key to be updated must exist in the database. To realize an update, we maintain a vector (denoted by V) that stores the same set of elements of HS (a hash set does not have $O(1)$ complexity of random access, so we use a vector here). We then randomly pick one key from the vector V, generate a *value* for the key, and write the pair to the output. The number of updates is specified by `-U` in the command line.

Point Query (PQ). Point queries (PQs) look for a key in the database and returns the corresponding *value* if the key exists. Note that, in practice, a target key may or may not exist in the database. To differentiate between empty and non-empty PQs, we use the information stored in a hash set HS and a vector V. To generate an empty PQ, we first randomly generate a key k, and then we check if $k \in HS$. If it does, we re-generate a key at random; otherwise, we add a point query on k to the output. To generate a non-empty PQ, we simply pick a key at random from the vector V. KVBench allows the users to specify the exact proportion of empty and non-empty PQs, and thereby, emulate a new class of real-life workloads. Users can use the two command line arguments, `-Q` and `-Z`, which specify the total number of PQs in a workload and the fraction of empty PQs, respectively. By definition, a valid value for `-Z` should be a floating point number in $[0, 1]$.

Range Query (RQ). Range queries return all entries with keys that fall within a specified range (i.e., a start key and an end key). The size of the result set depends on the *selectivity* of the range query. The selectivity of a range query (denoted by Y) refers to the fraction between the expected cardinality of the result-set returned and the number of unique and existing key-value pairs in the database. In practice, the performance of a data store can vary widely with the selectivity of range queries, and often, query latency varies, non-linearly with the selectivity of the query. Thus, it is important for a workload generator to be able to generate range queries with a specific selectivity, as opposed to supporting queries with random selectivity only. To achieve this, we sort the vector V, randomly pick an index in $[0, \lfloor \|V\| \cdot (1 - Y) \rfloor]$ as the start key, and obtain the index of the end key by right-shifting the start index by $Y \cdot \|V\|$. In the command line, users can set the number of range queries using `-S` and selectivity using `-Y`. By definition, a valid value for `-Y` should be a floating point number in $(0, 1]$.

Point Delete. This operation is used to remove or invalidate an existing entry in the database. We implement this by randomly

picking an index in the vector V and deleting it from both V and the hash set HS. This ensures future operations on existing keys (i.e., updates and non-empty PQs) cannot be triggered for the deleted key before it is inserted again. The number of point deletes is specified by `-D` in the command line.

Range Delete. A range delete removes or invalidates all entries that fall within a given range. Similarly to range queries, KVBench can specify the selectivity of a range delete (denoted by y). However, to ensure correctness, after a range delete is generated, we remove all the entries within the range from both V and HS. In the command line, `-R` and `-y` are used to specify the number of range deletes and the associated selectivity, respectively. The selectivity of a range delete (`-y`) is a floating point number in $(0, 1]$.

3.2 Distributions for Skewed Workloads

Key-value workloads, in practice, often exhibit skewed access patterns [6]. While state-of-the-art key-value workload generators commonly support various parameterized distributions, they are unable to generate workloads where different operations follow different distributions. This is a significant limitation of all state-of-the-art workload generators, and as a result, they are unable to emulate a large set of real-world workloads. KVBench allows the users to specify the distributions individually for every point operation, including insert, update, and point query, as shown in Figure 1(A). For example, KVBench can generate an interleaved workload where updates follow a uniform distribution, but PQs (point queries) follow some skewed distribution (say, Zipfian). In this section, we discuss all supported distributions in KVBench and explain how they are integrated with basic key-value operations.

3.2.1 Distributions Supported. We summarize the distributions supported in KVBench along with their input parameters in Table 3. Below, we discuss the implementation of each distribution.

Uniform. For uniform distribution, the key for an operation is chosen within a specific domain, using `uniform_int_distribution()` in C++. For updates and non-empty PQs, the key is chosen from a randomly generated index in vector V.

Normal. A normal distribution is parameterized by a mean μ and a standard deviation σ. In KVBench, we use the mean percentile to indicate the index position of the mean value and a scaled standard deviation. For example, when the mean percentile and the standard deviation for updates are specified as p_U and σ_U, respectively, the index of the key to be updated in sorted V (noted by $ikey$) should follow $\mathcal{N}(\mu, \sigma^2)$ where $\mu = p_U \cdot \|V\|$ and $\sigma = \sigma_U \cdot \|V\|$.

Beta. The Beta distribution is a continuous probability distribution defined on the interval $[0, 1]$ and parameterized by two positive parameters α and β. Similar to the normal distribution, we use $Beta(\alpha, \beta)$ to generate a random index $ikey$ within $[0, \|V\| - 1]$ and choose the key $V[ikey]$ as the operation object. In the implementation, we use two gamma distributions to simulate a beta distribution. Specifically, we use two additional random variables a, b where $a \sim \Gamma(\alpha, 1)$, $b \sim \Gamma(\beta, 1)$, and then we obtain the index of the selected key by $\left\lfloor \frac{a \cdot \|V\|}{a+b} \right\rfloor$, where $\frac{a}{a+b} \sim Beta(\alpha, \beta)$.

Zipfian. A Zipfian distribution is a skewed distribution that is typically characterized by the parameter α_Z. Unlike normal and beta distributions, a Zipfian distribution does not specify what the selected probability is for a given key in a finite key space. Instead, the Zipfian distribution specifies the relationship between the frequency ranking of a key and its probability of being selected. Thus, to emulate a Zipfian distribution, we need to shuffle the entries in the vector V to generate a new vector V', which is only used within the Zipfian-based generator. After shuffling, we then apply the Zipfian distribution to randomly select an index in vector V' where the probability the i^{th} element (starting from 0) in V' is chosen is $\frac{1/(i+1)^{\alpha_Z}}{\sum_{j=1}^{\|V\|} 1/j^{\alpha_Z}}$.

3.2.2 Key-Value Operations that Support Distributions. As shown in Table 3, KVBench supports inserts, updates, and empty and non-empty PQs to have any of the above distributions.

Inserts. As KVBench distinguishes unique inserts from updates, when generating (unique) inserts with a specific distribution, we set the key prefix to follow the specified distribution, while the remaining part of the key is generated randomly. Inserting unique keys with a specific distribution is part of several real-life workloads. While YCSB does not support this feature, db_bench allows the prefix of inserted keys to follow a parameterized two-term-exponential distribution. This knob helps db_bench achieve a more accurate simulation of real-life workloads (e.g., ZippyDB, an internal workload at Meta) [6]. To implement this, we can enumerate all possible prefixes and construct a vector V_{pre} to store them, for a given prefix length. For integer keys, KVBench uses the first 10 bits as the prefix while for string keys, the first two characters are used. As such, we have $\|V_{pre}\| = 2^{10} = 1024$ for integer keys and $\|V_{pre}\| = 62^2 = 3844$ for string keys (the number of valid characters is 62 in KVBench). We can then emulate a distribution within vector V to obtain a prefix and randomly produce the padding characters or bits when generating a key to be inserted.

Updates and Non-Empty Point Queries. A skewed distribution for updates or non-empty PQs indicates that a small set of keys in vector V are more frequently updated or queried than the others. As we maintain the vector V that stores all the existing keys in the database, to emulate a specific distribution for updates and non-empty PQs, we simply need to repeatedly pick an index within $[0, \|V\| - 1]$ according to the user-specified distribution.

Empty Point Queries. It is impossible (especially, for string keys) to enumerate all keys that are not in the vector V and construct another key space to generate empty PQs. Thus, for practical purposes, we construct a key set HS_{zero} (hash set) and V_{zero} (vector) in advance, specifically for empty PQs (i.e., before generating inserts). The size of the hash set HS_{zero} can be customized by a command line argument `--UZ`, which specifies the fraction between the unique empty PQs and the total number of empty PQs. Specifically, we have $\|V_{zero}\| = \texttt{UZ} \cdot \texttt{Z} \cdot \texttt{Q}$ where `Z` is the proportion of empty queries in all the PQs, and `Q` is the specified number of PQs. After we have a finite and small key space in vector V_{zero}, we can use the same method as before to produce skewed empty PQs.

3.3 Mixed Workloads

Practical workloads often interleave different types of operations. A typical implementation of this in state-of-the-art is to generate a random key-value operation uniformly among a set of predefined key-value operations. For example, db_bench randomly picks

Table 3: Distribution specification in KVBench. The operation type `X` can be any from {`U`, `Z`, `E`, `I`} where `U`, `Z`, `E` represent updates, empty point queries, non-empty point queries, respectively, and `I` denotes the distribution of prefix of inserted keys.

args	description	range of values; default
`--[X]D`	distribution type of the operation type `X`	{0: uniform, 1:normal, 2:beta, 3:zipfian}; 0
`--[X]D_NMP`	mean percentile of the normal distribution of `X`	[0,1]; 0.5
`--[X]D_NDEV`	standard deviation of the normal distribution of `X`	(0,DOUBLE_MAX]; 1.0
`--[X]D_BALPHA`; `--[X]D_BBETA`	α and β, respectively, of the beta distribution of `X`	(0,1]; 1.0
`--[X]D_ZALPHA`	α of the Zipfian distribution of `X`	[0,DOUBLE_MAX]; 1.0

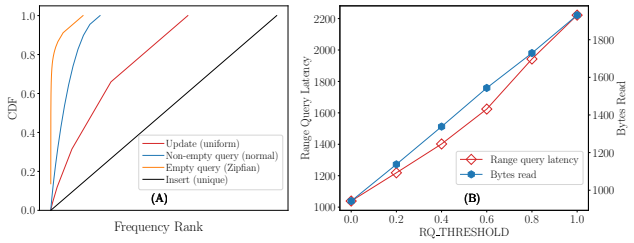

Figure 1: (A) A KVBench mixed workload with 100K uniform updates, 100K non-empty PQs with normal distribution ($p_U = 0.2, \sigma_U = 3.0$), 100$K$ empty PQs with Zipfian distribution ($\alpha_Z = 1.1$), and 100K unique inserts. (B) Different RQ_THRESHOLD with the same range query selectivity has very different performance.

an operation type from point queries, range queries, and writes (db_bench generalizes inserts and updates to writes) according to the specified proportion of each type of operation. KVBench, on the other hand, offers a richer set of knobs to control how to merge different types of operations into one workload.

Threshold-based Interleaved Workloads. KVBench uses five threshold parameters to determine the fraction of inserts before any other type of operation, as shown in Table 4. For example, when we set `-I1000000`, PQ_THRESHOLD = 0.3 and U_THRESHOLD = 0.5, PQs can only be generated after the first 300K inserts and updates can only be generated after the first 500K inserts. We highlight that these thresholds can have a significant performance impact. For instance, we generate a mixed workload with 4M inserts and 1K range queries with different RQ_THRESHOLD (the fraction of inserts before the first range query). When the range queries have the same selectivity, range queries after more data is ingested have to read more data from disk. As such, higher RQ_THRESHOLD leads to higher query latency, as shown in Figure 1(B).

Deletes/Inserts in Interleaved Workloads. Generating an interleaved workload with deletes/inserts is not trivial. When inserts and deletes are interleaved with other operations, the key generators have to be updated accordingly since the entries in vector V are not fixed. Updating the key generators adds up non-negligible overhead because we need to ensure that the vector V is sorted whenever a new key is inserted or an existing key is deleted (V has to be sorted to support range queries and range deletes). Although V does not need to be sorted for Zipfian distribution, deleting or adding a (new) key leads to recalculating the probability of generating each key, which increases the maintenance overhead. Despite this, KVBench efficiently generates workload with interleaved deletes, inserts, and other operations that follow a uniform distribution.

Preloading: A Special Case of Mixed Workload. KVBench provides the feature that the workload can be populated based on an existing insert-only workload. KVBench reads all the inserts in the input workload file to construct the hash set HS and the vector V. The remaining execution is the same as the original workload.

Table 4: The macro thresholds for interleaved workloads.

macros	description
U_THRESHOLD	fraction of inserts before updates
PD_THRESHOLD	fraction of inserts before point deletes
RD_THRESHOLD	fraction of inserts before range deletes
PQ_THRESHOLD	fraction of inserts before point queries
RQ_THRESHOLD	fraction of inserts before range queries

3.4 Emulating YCSB Workloads

KVBench is a generic key-value benchmarking suite that can emulate many workloads in YCSB. Take workload YCSB-A as an example. By default, workload A in YCSB populates a database where the preloaded number of key-value pairs is specified by `recordcount`, and it benchmarks the performance when executing a workload consisting of 50% reads and 50% updates (the total number of operations is specified by `operationcount`). In KVBench, we can use `-I` to produce the dataset used to populate the database, and then we respectively specify `-Q` and `-U` for reads and writes by preloading the insert-only workload (since `CoreWorkload` in YCSB only supports non-empty queries, we should also set `-Z` to 0 to ensure that all PQs are restricted to non-empty queries).

4 BENCHMARK WORKLOADS

KVBench can produce synthetic workloads that resemble real-life application workloads. This empowers NoSQL users to verify the performance of their system against various use cases. In this section, we introduce a new set of benchmark workloads and their usage in real-world applications that can be generated using KVBench.

KVBench-I: Empty Point Query-Oriented with Preloading. This workload consists of 100% PQs issued to an existing database, populated by an insert-only workload. The proportion of empty PQs is 80%. Empty PQs follow a beta distribution and the non-empty PQs follow a uniform distribution. To produce this workload, we first generate an insert-only workload, and then we use the preloading feature to generate a point-query-only workload by specifying the number of PQs, and the fraction and distribution of empty PQs.

Example: Customer inquiries for products follow this pattern when popular products are not in inventory. YouTube or Spotify searches without proper pattern matching, searching for books in a library catalog available for checkout, and searching for available doctor's appointments all fall under this category.

KVBench-II: Insert, Delete, Query, Update Interleaved. This workload consists of interleaving inserts (50%), deletes (10%), empty

Table 5: Benchmark workloads supported by KVBench (PQ stands for point queries)

workload type	workload composition (%)	distribution of operations
KVBench-I: Empty PQ-heavy (preloading)	20% non-empty PQ, 80% empty PQ	ZD = beta, ED = uniform
KVBench-II: Interleaved inserts, deletes, PQs, updates	50% insert, 10% delete, 15% empty PQ, 25% update	ID = ED = UD = uniform
KVBench-III: Multi-distribution update and PQ (preloading)	50% updates, 25% non-empty PQ, 25% empty PQ	UD = Zipfian, ED = ZD = uniform
KVBench-IV: Update and range delete-heavy (preloading)	50% updates, 50% range delete	UD = Zipfian
KVBench-V: Insert-heavy with skewed prefixes	95% insert, 5% non-empty PQ	ID = Zipfian, ZD = uniform

PQs (15%), and updates (25%). Inserts, empty PQs, and updates follow a uniform distribution, i.e., all keys are equally likely to be inserted, searched, updated, and deleted.

Example: Social media users frequently create, update, and delete their posts or search for posts. Online shoppers regularly add and remove products from their carts, search for specific products, etc.

KVBench-III: Multi-distribution Updates and PQs with Preloading. This workload consists of 50% updates, 25% empty PQs, and 25% non-empty PQs. Updates and non-empty PQs follow a Zipfian distribution, while empty PQs are uniformly distributed. Similar to the query-only workload, this workload is executed in an existing database. As such, we also use the preloading feature to generate this workload by specifying the distribution of updates and PQs.

Example: On social media platforms, users frequently update and search for existing posts on popular keywords and seldom search for non-existing keywords/posts/people. In scientific databases, there are frequent updates to citation counts and searches for popular landmark research papers while the search for absent papers occurs with low uniform probability.

KVBench-IV: Update and Range Delete Heavy with Preloading. This workload consists of 50% updates and 50% range deletes, the inserts are preloaded from the existing workload file. The updates follow Zipfian/beta distribution.

Example: In superstores or e-commerce platforms, products older than a certain timestamp are regularly deleted from current inventory when they are sold out. The price of items with low demand (stock greater than a certain value and older than a certain timestamp) is regularly updated and put to clearance.

KVBench-V: Insert-Heavy with Skewed Prefixes. This workload consists of 95% inserts and 5% uniform non-empty PQs. The prefixes of inserted keys follow a Zipfian distribution.

Example: Sensor data generated by IoT devices and energy consumption levels by households and businesses in a smart grid ecosystem can both be modeled using Zipfian distribution. Specifically, when the data collected by IoT uses timestamps as a prefix of key, we may have much more data collected from daytime instead of midnight since human activities and energy consumption are more active during daytime. As such, the prefix of inserted keys may follow a skewed distribution.

Uncovering New Performance Patterns. The key reason for developing KVBench is to shed light on previously uncovered system behavior. Using the developed workloads, we highlight two examples of new performance patterns observed with KVBench. Here, we experiment with RocksDB (v8.9.1), we use a PCIe P4510 SSD with direct I/O enabled, and we use 16MB for the block cache. We preload our database with 10M entries.

Figure 2(A) shows the query latency as we vary the fraction of empty PQs, Z, on a uniform and a Zipfian distribution. First, we see that as Z increases, the overall query latency drops precipitously because accesses to the raw data can be skipped via auxiliary structures like Bloom filters and Zonemaps, while non-empty queries always access at least one data page. Second, we observe that a Zipfian distribution leads to consistently faster queries because skewed queries lead to a higher cache hit rate.

Figure 2(B) shows that different interleavings of the same workload may lead to dramatically different system behavior. Here, we generate two workloads with $0.9M$ updates, $0.1M$ deletes, and $4M$ non-empty PQs. The "Sequential" workload executes updates, deletes, and PQs serially, while the "Interleaved" one mixes the operations. Despite having identical workload compositions, the two executions lead to very different behavior. Specifically, since RocksDB is write-optimized, the sequential workload has 1.6× higher throughput than the interleaved one, as long as it executes updates and deletes, while the two workloads perform very similarly for queries. We also highlight that, although the throughput of two workloads becomes similar as more queries are executed, a "Sequential" workload still has a marginal benefit over a "Interleaved" workload even after all operations are completed.

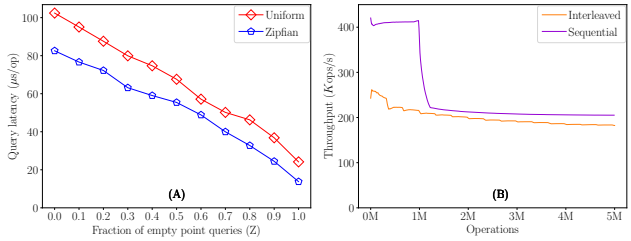

Figure 2: (A) The average latency per query decreases as there are more empty PQs for different distributions. (B) The throughput differs substantially between interleaved and sequential workloads even if they have the same workload composition.

5 CONCLUSION

In this paper, we present KVBench, a scalable and distribution-aware key-value workload generator for NoSQL systems. KVBench can better emulate real-life workloads by offering more knobs for new workload characteristics that are not supported by the state-of-the-art. Specifically, KVBench differentiates between empty and non-empty point queries, allows the distribution to vary across different basic key-value operation types, and further supports deletes in an interleaved workload. Future optimizations in NoSQL systems can benefit from KVBench when examining the performance impact of these new workload characteristics.

ACKNOWLEDGMENTS

This work was funded by NSF grant IIS-2144547, a Facebook Faculty Research Award & a Meta gift.

REFERENCES

[1] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Transactions on Computers (TC)* 65, 3 (2016), 902–915. https://doi.org/10.1109/TC.2015.2435779

[2] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).

[3] Apache. 2023. HBase. *http://hbase.apache.org/* (2023).

[4] 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 ACM SIGMOD International Conference on Management of Data*. 1185–1196. https://doi.org/10.1145/2463676.2465296

[5] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*. http://link.springer.com/chapter/10.1007%2F978-3-319-04936-6_5

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[7] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 275–290. https://doi.org/10.1145/3183713.3196898

[8] CockroachDB. 2021. CockroachDB. *https://github.com/cockroachdb/cockroach* (2021).

[9] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 143–154. https://doi.org/10.1145/1807128.1807152

[11] Couchbase. 2017. ForestDB Benchmark. *https://github.com/couchbaselabs/ForestDB-Benchmark* (2017).

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. https://doi.org/10.1145/3035918.3064054

[13] Facebook. 2024. db_bench. *https://github.com/facebook/rocksdb/wiki/Benchmarking-tools* (2024).

[14] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).

[15] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).

[16] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[17] Ren Jinglei, Kjellqvist Chris, and Deng Long. 2024. YCSB-C. (2024). https://github.com/basicthinker/YCSB-C

[18] Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights & Implications. In *International Workshop on Testing Database Systems (DBTest)*. https://doi.org/10.1145/3662165.3662766

[19] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[20] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230. https://doi.org/10.14778/3415478.3415546

[21] MongoDB. 2023. Online reference. *http://www.mongodb.com/* (2023).

[22] Oracle. 2024. Oracle NoSQL. *https://docs.oracle.com/database/nosql-12.1.3.0/GettingStartedGuideTables/tablesapi.html* (2024).

[23] Redis. 2024. Online reference. *http://redis.io/* (2024).

[24] Mohammed Suhail Rehman and Aaron J Elmore. 2022. FuzzyData: A Scalable Workload Generator for Testing Dataframe Workflow Systems. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 17–24. https://doi.org/10.1145/3531348.3532178

[25] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. 2021. OrientDB: A NoSQL, Open Source MMDMS. In *Proceedings of the The British International Conference on Databases 2021, London, United Kingdom, March 28, 2022 (CEUR Workshop Proceedings, Vol. 3163)*. 10–19. https://ceur-ws.org/Vol-3163/BICOD21_paper_3.pdf

[26] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2489–2497. https://doi.org/10.1145/3514221.3522563

[27] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432. https://doi.org/10.1145/3514221.3520169

[28] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM Design Space and its Read Optimizations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3578–3684. https://doi.org/10.1109/ICDE55515.2023.00273

[29] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757

[30] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2023. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems (TODS)* 48, 3 (2023), 8:1—-8:40. https://doi.org/10.1145/3599724

[31] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. https://doi.org/10.14778/3476249.3476274

[32] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. 2008. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, Victoria, BC, Canada, September 27, 2008*. 41–48. https://doi.org/10.1145/1411273.1411280

[33] TPC. 2021. TPC-H benchmark. *http://www.tpc.org/tpch/* (2021).

[34] WiredTiger. 2021. Source Code. *https://github.com/wiredtiger/wiredtiger* (2021).

[35] Zichen Zhu, Sarkar Sarkar, and Manos Athanassoulis. 2023. Acheron: Persisting Tombstones in LSM Engines. In *Companion of the International Conference on Management of Data (SIGMOD)*. 131–134. https://doi.org/10.1145/3555041.3589719