Tectonic: Bridging Synthetic and Real-World Workloads for Key-Value Benchmarking

Alexander H. Ott, Shubham Kaushik, Boao Chen, and Subhadeep Sarkar

Brandeis University, MA, USA
{alexanderott, kaushiks, boaochen, subhadeep}@brandeis.edu

Abstract. Key-value stores are the backbone of many modern SQL- and NoSQL-based data systems, serving a variety of real-world applications. Despite their widespread adoption, existing key-value benchmarks fall short across multiple dimensions when accurately replicating complex and dynamic real-world workloads. For instance, state-of-the-art key-value benchmarks, such as YCSB, KVBench, and db_bench, are unable to (i) emulate dynamic workloads where the workload composition and distribution changes arbitrarily over time; (ii) generate composite keys with different prefix distributions; and (iii) generate workloads with varied degrees of data sortedness. These limitations result in inaccurate performance evaluations and limit the ability to understand how a commercial key-value store performs under dynamically shifting workloads.

In this paper, we introduce Tectonic, a Rust-based, highly configurable, and resource-efficient key-value workload generator designed to model the temporal, structural, and dynamic properties of real-world workloads. Tectonic offers (i) fine-grained control over data access patterns for inserts, updates, merges, point and range queries, and point and range deletes; (ii) configurable composite key generation/selection strategies; (iii) dynamic workload generation where the workload properties change over time; and (iv) generation of workloads with user-specified data sortedness. Tectonic does so (v) at a 2× higher throughput than the state-of-the-art, (vi) while recording up to 84% lower main memory footprint. By bridging the gap between synthetic and production workloads, Tectonic enables in-depth analysis of key-value data systems under conditions that better reflect the demands of real-world applications. We benchmark Tectonic's performance against YCSB and KVBench in terms of latency, resource utilization, and ability to emulate production workloads. The code for Tectonic is available at: https://github.com/SSD-Brandeis/tectonic.

Keywords: Key-value benchmark · Dynamic workload · Scalability.

1 Introduction

Key-value stores are widely used as the storage engine of a wide number of NoSQL data stores, including RocksDB [15] at Meta, Bigtable [7] and LevelDB [18] at Google, FoundationDB [3] at Apple, Cassandra [1] and HBase [2] at

Apache, Azure CosmosDB [26] and FASTER [6] at Microsoft, and DynamoDB [12] at Amazon. Many relational databases also employ key-value stores underneath in the storage layer. For example, CockroachDB [10] uses Pebble as the storage engine, and TiDB [19] and MyRocks [13] both use RocksDB as the underlying storage. While the key-value paradigm allows for low-latency writes and high operational throughput in general, the performance of a key-value store can vary significantly based on the workload characteristics, such as the proportion of different operations, the distribution of keys, and the data access patterns, as well as the variability of the workload over time [5,8,22,24,35]. Thus, there is a strong need for effective benchmarking tools that can (i) generate workloads that closely resemble production workloads and (ii) evaluate the performance of key-value stores under diverse and dynamic workloads.

Limitations of the State of the Art. State-of-the-art key-value benchmarks, such as YCSB [11], KVBench [37], and db_bench [14], are widely used to evaluate the performance of key-value stores. However, these workload generators are **unable to capture the complexity of real-world workloads across multiple dimensions**, as shown in Table 1.

(1) Inability to Generate Dynamic Workloads: In practice, production workloads often exhibit temporal variations, such as diurnal shifts, seasonal trends, and gradual transitions in access patterns [5,21,37]. For instance, over a 7-day window production workloads at Meta, such as UDB, ZippyDB, and UP2X, vary across multiple dimensions, including (i) the composition of the workload, (ii) the distribution of the operations, (iii) access patterns of queries, as well as (iv) the sizes of key-value pairs [5]. UDB shows a diurnal pattern for writes to the database, which peaks each day at 17:00 PST, and then quickly reaches its nadir at 23:00 PST. Moreover, during the weekends, some column families experience a significant (by up to 10×) jump in the query frequency. One of the biggest limitations of state-of-the-art key-value benchmarks is their **inability to capture dynamic variations in the workload altogether and model shifting workloads**. In fact, the only way to emulate temporal variation in workloads is by generating segments of the dynamic workload separately and then stitching them together manually. This is a very crude emulation of dynamic workloads, as each phase operates on a disjoint keyspace, thus failing to preserve key-sharing across phases. As a result, the workloads generated lack coherence, which in turn, undermines the accuracy of performance evaluation.

(2) No Support for Composite Keys and Prefix-based Access Patterns: In production workloads, often the database name, the relation/column family name, as well as the user ID are added as a prefix to the *actual* application-generated key to constitute a **synthetic composite key**. Prefixes to this composite key are used as identifiers for a key-value pair's origin and domain, enabling efficient filtering and grouping. For instance, a composite key `cf-EMP:edge:alex` indicates the key `alex` was generated by an application `edge` and is stored in the column family `EMP` [31]. Existing key-value benchmarks are unable to model prefix-based key layouts altogether. Generating composite key in YCBS, db_bench, or KVBench would entail significant manual development, which is both complex

Table 1: Tectonic supports a wide range of operations, distributions, and structural controls, enabling accurate modeling of real-life workloads.

		YCSB	KVBench	db_bench	Tectonic
operations	insert	✓	✓	✓	✓
	update	✓	✓	✓	✓
	read-modify-write	✓		✓	✓
	point query	✓	✓	✓	✓
	empty point query		✓	✓	✓
	range query	✓	✓	✓	✓
	point delete		✓	✓	✓
	empty point delete		✓		✓
	range delete		✓	✓	✓
distributions	uniform	✓	✓	✓	✓
	normal		✓	✓	✓
	beta		✓		✓
	zipfian	✓	✓		✓
	exponential			✓	✓
	log normal				✓
	poisson				✓
	weibull				✓
	pareto			✓	✓
properties	dynamic workload shifts		*	*	✓
	context-aware shifting				✓
	data sortedness				✓
	variable query selectivity	✓		✓	✓
	variable key-value length				✓
	temporality-based access			✓	✓
	customizable key prefix			*	✓
	composite keys				✓

* requires significant manual intervention

and time-consuming. This makes it very hard to evaluate the performance of key-value stores that use prefix-based operations, such as indexing [17], membership checks [32], and data layout reorganization [23].

(3) Lack of Sortedness Benchmarks: Production workloads often exhibit a correlation between attributes such that when the entries are sorted on one attribute, they are also *nearly sorted* on another [29,30]. For instance, in TPC-H [36], the `lineitem` table has three date columns, namely, `shipdate`, `commitdate`, and `receiptdate`, and sorting the data on `shipdate` would cause the data to be very close to being sorted on `commitdate` and `receiptdate` as well (but not fully-sorted) [4]. Sortedness of data affects the ingestion, indexing, and query cost significantly, and benchmarking the performance of a storage engine under varied data sortedness is crucial in practice. While existing key-value benchmarks can generate data following a number of key distributions [14,37], they are **unable to generate data with variable sortedness**. Benchmark on Data

Sortedness (BoDS) [28] remains the only sortedness benchmark available in the literature; however, it has two major limitations. First, it is not designed as a general-purpose key-value workload generator, as it does not support basic operations, such as deletes and updates, cannot generate interleaved operations, and does not support different key distributions beyond sortedness. Second, BoDS supports only numeric key generation, and the keys are always generated serially (and not randomly).

(4) Lack of a Unified Benchmarking Framework: Modern application requirements are heterogeneous, exhibiting diverse and dynamic data access patterns [5,25,27]. It is important for a database practitioner to benchmark the performance of different key-value stores under diverse and shifting workloads to ascertain the suitability of a database for a given workload and performance target. Existing key-value benchmarks, however, **fall short in emulating the rich space of key-value workloads** along one dimension or another. For instance, YCSB is limited to a fixed set of distributions (i.e, uniform and Zipfian) and can not generate workloads with point or range deletes [11,37]. db_bench is tightly coupled with RocksDB and LevelDB only and, hence, has a limited applicability for other key-value stores [14]. KVBench does not support read-modify-write operations and has a remarkably high memory footprint (we discuss this in Section 3).

Due to these limitations, state-of-the-art key-value benchmarks are **unable to accurately replicate real-world workloads, leading to poor benchmarking and suboptimal data system optimizations**.

Contributions. In this paper, we present Tectonic, a Rust-based, scalable, and highly configurable key-value benchmarking suite that can generate a diverse array of dynamic workloads while being faster and more resource-efficient than the state of the art. The contributions of Tectonic are as follows.

1. *Dynamic and Evolving Workloads*: Tectonic exposes a rich set of tunable knobs, providing the users fine-grained control over the workload generation process. Tectonic allows the user to model dynamically shifting workloads by specifying the workload characteristics for each phase. Each phase is independently configurable, allowing the keyspace to be either disjoint or shared according to the application requirements.
2. *Structured Key Generation*: Tectonic allows configurable composite key generation, allowing the users to have control over the size and domain of each key segment. Each key segment may have distinct distributions, which allows modeling the "hotness" of key range, as observed in production workloads.
3. *Sortedness-Aware Ingestion*: Tectonic supports generating entries with varied degrees of sortedness. Unlike the state of the art [28], entries in Tectonic can be generated as interleaved with other database operations and following any key distribution, while following a user-specified sortedness.
4. *Fast and Resource Efficient*: We show through experiments that the Rust-based implementation of Tectonic makes it significantly faster than state-of-the-art workload generators, such as YCSB and KVBench, while recording an, on average, 7.5× lower memory footprint.

2 The Tectonic Benchmark Suite

Tectonic is a highly configurable and resource-aware key-value benchmark that supports a diverse set of operations, distributions, and properties (see Table 1) to ensure accurate emulation of real-world workloads. Tectonic's Rust-based codebase allows for orders of magnitude faster workload generation (by up to 12×) with a significantly smaller main memory footprint (by up to 84%) and comparable CPU utilization. To the best of our knowledge, **Tectonic is the first key-value benchmark that supports benchmarking storage engines under dynamically shifting workloads**. Users can specify the workload specification for every phase of a dynamic workload through a simple JSON configuration file, which Tectonic uses as input.

2.1 Architecture and Overview of Tectonic

Execution Pipeline. Benchmarking in Tectonic happens over a five-step process as shown in Fig. 1.

Step ①: Tectonic takes as input a JSON file, which contains the specification for the benchmarking workload. At the top level of the specification, we have a list of one or more *Section*. A *Section* is defined as an ensemble of *Groups* of operations. Each *Group*, in turn, is a collection of an arbitrary number of insert, update, empty and non-empty point query, range query, read-modify-write, and point and range delete operations. In the resulting workload, the operations in different *Sections* and *Groups* within a *Section* are executed serially. Operations within the same *Group* are interleaved.

Step ②: Tectonic parses the JSON input and converts it to a structure holding *Sections*, the containing *Groups*, and the distribution details necessary to generate the workload. At this stage, Tectonic identifies the different phases in the benchmarking setup by analyzing the *Sections* and *Groups* in the input file. Operations in the same *Group* are (i) to be executed as interleaved, (ii) share the same keyspace, and (iii) are affected by preceding operations within the same group. For every *Section* in the workload, the generator in Tectonic performs the following steps.

Step ③: The generator chooses the appropriate data structure to create an *ActiveKeySet* that stores the **set of active keys** at any point during workload generation. The *ActiveKeySet* is a critical component of the generator as it allows for (i) differentiating between unique inserts and updates, (ii) differentiating between empty and non-empty operations (i.e., point queries and deletes), (iii) supporting a wide array of access distributions, and (iv) generating data with varied degrees of sortedness. Based on the specification of the operations in a *Section*, Tectonic chooses the appropriate data structure (vector, hash set, Bloom filter, hash-map, or B^+-tree) to implement the *ActiveKeySet*. Each *Section* operates on its own independent *ActiveKeySet*, allowing for *Sections* to be generated in parallel.

Steps ④ & ⑤: Next, Tectonic performs the generation (from unique inserts) and selection (which key to query) phases and passes the output to the writer for

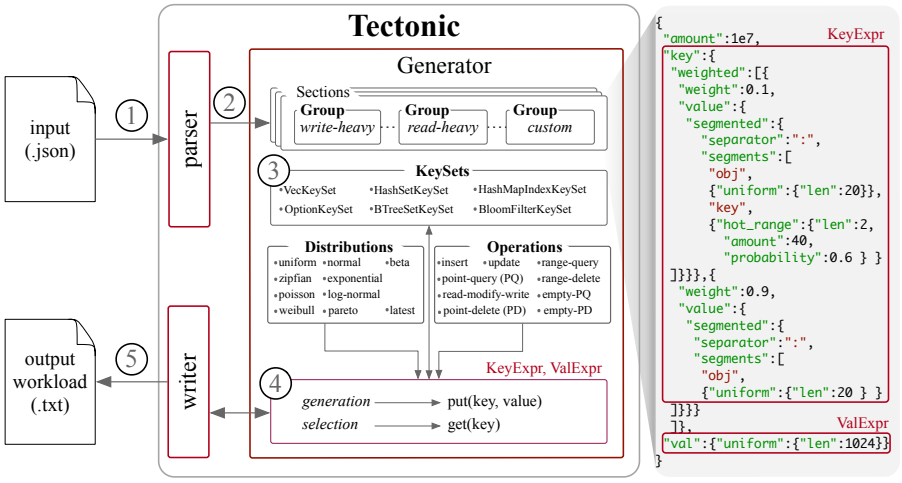

Fig. 1: The components of Tectonic and a flow for generating workloads.

execution (or persistent storage). For inserts, updates, and read-modify-writes, the values associated with the keys are generated on the fly.

Key Construction. Tectonic constructs keys based on the specification in the input file. A `NumberExpr` (`NumExp`) defines a numeric expression that evaluates to a numeric key. A `StringExp` (`StrExp`), similar to `NumExp`, is an expression that evaluates to the desired string key. `StrExp` has five variants: (i) `constant`, evaluates to a constant string, (ii) `uniform`, produces a string of a specific length composed of uniform characters (defaulting to alphanumeric), (iii) `weighted`, randomly selects from an array of `StrExp` according to configured probability weights, (iv) `segmented`, evaluates different segments independently based on a configured distribution and joins them together using a delimiter to form a complete key, and (v) `hot-range`, generates certain prefixes more frequently.

The `NumExp` is used by both key generation and key selection. During key generation, it controls the parameters such as the lengths of the keys and values, as well as the selection of hot key ranges. The selection process selects keys from the existing *ActiveKeySet* for operations such as updates, read-modify-write, point queries, range queries, point deletes, and range deletes. Keys are selected by sampling by a specified distribution (defaults to uniform $[0.0, 1.0)$), clamping the value to $[0.0, 1.0)$, and multiplying the total count of keys to calculate the index, which is used to retrieve the key from the *ActiveKeySet*.

2.2 Basic Key-Value Operations

We now describe the implementation of core key-value operations in Tectonic.

Insert. Insert in Tectonic adds a key-value pair with a **unique key** to the database. Tectonic maintains the *ActiveKeySet* (KS) to store and track all active keys of a workload at any point in time. The *value* for a key is generated on

the fly by the writer when the record is written to the output file. Users can configure the insert workload by defining a `"inserts"` block with `StrExp` for `"key"` and `"val"` within `"groups"` as: `{"groups":[{"inserts":{"amount":NumExp, "key":StrExp,"val":StrExp}}]}`.

Updates and Read-Modify-Writes. Updates overwrite the value of an existing key-value pair in the database, while read-modify-writes (also known as *merge*) append a new value to the existing one or run a user-defined function to merge the values [11,15]. In both cases, the key must already exist in the database and must not be logically invalidated. When generating an update or a read-modify-write, Tectonic selects a key from the *ActiveKeySet*, KS container after sampling based on the distribution (`Dist`) specified by the user in `"selection"`. The default distribution is: `Uniform(0, 1)`. It then generates a value to be updated or merged while writing keys to the workload file, similarly to inserts. Users must pass an `"updates"` and `"merges"` key to generate update and merge operations, respectively, as: `{"groups":[{"updates":{"amount":NumExp,"val":StrExp, "selection":Dist}}]}`.

Point Query. Tectonic can generate point queries (PQs) that are (i) *empty*, i.e., when the target key does not exist in the database or has been logically invalidated, or (ii) *non-empty*, i.e., when the target key is live. To differentiate between empty and non-empty PQ, Tectonic uses the metadata in KS. To generate an empty PQ, the generator first randomly generates a key k, and then checks if $k \in KS$. If it is, Tectonic re-generate a key at random; otherwise, it adds a PQ on k to the output. For non-empty PQs, Tectonic selects a key from the KS after sampling it on the distribution provided in the input file. Users must specify `"point_queries"` or `"empty_point_queries"` key to generate the corresponding operations as: `{"groups":[{"point_queries":{"amount":NumExp,"selection":Dist}` and `{"empty_point_queries":{"amount":NumExp,"key":StrExp}]}`.

Range Query. To generate range queries (RQs), Tectonic first probes KS and selects a start key by sampling based on a specified distribution. Then, the end key is determined based on the query selectivity. The RQ selectivity in Tectonic can be specified as constant or to follow a user-specified distribution. To generate RQs, the input JSON should have `"range_queries"` as: `{"groups":[{"range_queries":{"amount":NumExp,"selectivity":NumExp, "selection":Dist}]}`.

Point Delete. Tectonic can distinguish between *empty* and *non-empty* point deletes [33,34]. Similarly to PQs, to distinguish between empty and non-empty PDs, Tectonic probes KS for membership checks. For non-empty PDs, Tectonic selects a key from KS, and once the delete is generated, the key is removed from KS. Users can generate `"point_deletes"` or `"empty_point_deletes"`, by specifying: `{"groups":[{"point_deletes":{"amoun"t:NumExp},{"empty_point_deletes":{ "amount":NumExp,"key":StrExp}]}`.

Range Delete. Similar to RQs, Tectonic can generate range deletes with a given selectivity. After generating the RQ, Tectonic removes all keys within the deleted range from KS. User can generate `"range_deletes"`, as follows: `{"groups":[{"range_deletes":{"amount":NumExp,"selectivity":NumExp}}]}`.

2.3 Modeling Workload Skews

Tectonic supports nine distributions for key generation and selection: (1) uniform, (2) normal, (3) beta, (4) Zipfian, (5) exponential, (6) log-normal, (7) Poisson, (8) Weibull, and (9) Pareto. It leverages the `rand_distr` Rust library, which readily provides the implementations of the distributions, enabling fast sampling. Tectonic uses these distributions to sample keys from the `ActiveKeySet` and clamp them to the interval $[0.0, 1.0)$, to ensure validity.

Each operation in Tectonic, such as inserts, updates, point queries, or range deletes, can have their independent key distribution. This allows for modeling diverse access patterns across operations, such as uniformly distributed inserts followed by a workload of skewed (e.g., Zipfian) updates, or range queries with a uniform selectivity interleaved with non-empty point queries on the recently inserted (beta distribution) entries.

2.4 What Sets Tectonic Apart

Ability of Generating Dynamic Workloads. Production workloads are often non-uniform and evolve over time due to changes in user behavior, application requirements, or specific periodic events. Tectonic is designed to support the generation of dynamically shifting workloads as it allows the user to separately specify the composition of multiple workload phases (through *Groups* and *Sections* in the JSON file). Each phase has its own access distribution, operation mix, and key selection policy. Below, we present two real-world workloads that state-of-the-art key-value benchmarks are unable to produce, and we discuss how Tectonic is able to emulate them.

```
1 {
2  "groups":[{
3   "inserts":{
4    "amount":1e6,
5     "key":{"uniform":{"len":32}},
6     "val":{"uniform":{"len":1024}}
7   },
8   "updates":{
9    "amount":1e4,
10   "val":{"uniform":{"len":1024}}
11  }
12 },
```

```
13 {
14  "point_queries":{
15   "amount":1e3,
16   "selection":{"uniform":{"len":32}}
17  },
18  "range_queries":{
19   "amount":50,
20   "selectivity":{
21    "uniform":{"min":0.01,"max":0.1}},
22   "selection":{"uniform":{"len":32}}
23  }
24 }]}
```

Fig. 2: Dynamic workload specification in Tectonic: Phase 1: (Lines 1-12) for a write-heavy workload; Phase 2: (Lines 13-24) for a read-heavy workload.

Example 1: Production workloads at Meta and X experience a *burst of inserts and updates* during peak hours (i.e., during sports matches, political debates, etc.) [9], when user activity soars high [5]. The workload pattern during these hours is distinctively different from other times of a day, when users are primarily browsing content, leading to a *high volume of point and range queries* on popular or recently updated content. Note that the point and range queries in the later phase must share the keyspace with the preceding inserts and updates.

To simulate this workload scenario, Tectonic requires two groups in a section, one which can simulate bursty inserts and updates (Fig. 2 (Lines 1-12)) where key size is $32B$ and value is of $1024B$, two which can simulate high volume of point and range queries (Fig. 2 (Lines 13-24)) where both point queries and range queries follows a uniform distribution.

Example 2: E-commerce platforms, such as Amazon or Shopify, often process workloads where different phases focus on separate segments of the keyspace. For instance, during off-peak hours, a bulk of the operations target the product catalog, such as *filtering products by vendors*. This leads to skewed accesses to data with a specific key-prefix, such as `product:<vendor_id>:<sku>`. On the other hand, during peak hours when customers are active, operations such as *checking a user's order history* leads to frequent accesses use a completely different set of keys with a prefix, like `order_history:<user_id>:<timestamp>`. Since the keys across these phases are from disjoint parts of the keyspace, Tectonic model this workload by modeling each phase in a separate *Section*, thereby enabling distinct prefix layouts, key distributions, and operation interleaving for each phase.

Tunable Sortedness for Keys. Tectonic can generate workloads with varied degrees of sortedness. In several production workloads, data arrives in nearly sorted order [30,36], but not fully sorted. Tectonic, like BoDS, uses the K, L-sortedness metric to generate a collection of data that is arbitrarily sorted. K denotes the fraction of entries that are out of place in the workload, and L denotes the maximum displacement of such an out-of-place entry. However, unlike BoDS, Tectonic (i) can support string keys as well as composite keys, (ii) generates the data following a specific distribution, and (iii) allows other database operations to be interleaved with the inserts. This makes simulating and benchmarking real-life near-sorted workloads ingestion scenarios.

Emulating "Hot" Key-ranges. Similarly to UDB, ZippyDB, and UP2X, many production workloads exhibit strong skew in access patterns, where a small subset of keys, known as "hot" keys, are accessed far more frequently than others [5,20]. These hot regions may shift over time or remain persistent, depending on application behavior. For instance, during flash sales in e-commerce platforms like Amazon, only a few products, such as limited-time electronics or trending fashion items, receive the majority of user queries and orders. These product IDs form hot key ranges that dominate the workload for a short window. Tectonic supports fine-grained control over key selection by allowing users to define hot key ranges and assign access probabilities to them. The range for hot key ranges is defined using non-uniformly generated prefixes. The length of the prefix, the number of hot prefixes, and the probability of selecting a hot prefix over a non-hot one during string generation are specified.

Support for Prefix-based and Composite Key. The key-value stores often rely on structured keys to encode metadata, such as relation names or user types, for more efficient data partitioning and access. In practice, key-value stores such as RocksDB and Cassandra frequently use prefix-based or composite keys, e.g., `user:12345:profile` or `org:abc:project:xyz:log`, to co-locate related records and enable fast prefix scans. These structured keys are crucial for

building features such as fine-grained isolation (per relation) or logical grouping. Tectonic support prefix-based and composite key generation, allowing users to configure multiple segments with different distributions, or hot ranges. When describing the composition of a key, users can specify (i) the key length, (ii) how to construct each key segment, and (ii) the distribution of each segment.

3 Benchmark Workloads

Tectonic uses a JSON specification file as input to generate synthetic workloads that closely resemble real-life production workload characteristics. This input file is human-readable and is easy to configure. In this section, we show Tectonic can not only efficiently generate the benchmark workloads of YCSB and KVBench, but also generate a large set of workloads that the state of the art is unable to generate. We then compare Tectonic's performance against the state of the art in terms of end-to-end latency, throughput, and memory footprint.

Define Workloads in JSON. Each workload in Tectonic is described by a JSON file. The top-level structure contains *Sections*, a list of isolated phases in the workload, where operations in different sections do not share keyspace. The second level contains *Groups*, a list of sequential phases, where operations are aware of the previous phase (e.g., preloading followed by querying phase on the same keyspace). Each group specifies one or more operations (always interleaved), each with its properties. The most common one is *amount*, which refers to the number of operations to generate for a specific operation type. The *key/val* specifies the process for generating keys and values, e.g., uniformly random, segmented with a hot prefix, or composite keys. A workload can have multiple sections, with each section comprising various groups, and each group can have multiple types of operations.

Experimental Setup. We use a machine with an Intel(R) Xeon(R) Gold 6240R CPU, 2.40GHz cores, 192GB of RAM, 240GB SSD, and running Ubuntu 20.04 LTS. We use the GNU time utility (version 1.7) [16] to collect all system-level performance metrics such as user time, system time, and maximum resident set size. We simulate YCSB and KVBench workloads through Tectonic and compare the total execution time, CPU usage, and memory footprint.

3.1 Emulating YCSB Workloads

The YCSB key-value benchmark is widely used to evaluate the performance of key-value and NoSQL systems. It has six benchmark workloads (YCSB-A - YCSB-F), each representing a specific access pattern and read-write ratio. Tectonic can natively simulate all YCSB workloads using a simple JSON specification. To run a specific YCSB workload using Tectonic, the user provides a corresponding specification file (e.g., a.spec.json for YCSB-A) and executes: `tectonic-cli generate -w ycsb-specs/a.spec.json`. Each specification file encodes the YCSB workload logic using sections, groups, and operations, while maintaining consistency with the original YCSB design in terms of data model

and access patterns. We include full specifications for all six YCSB workloads as part of the Tectonic repository under ycsb-specs/, making it easy to modify, extend, or compose them with additional workload phases.

3.2 Emulating KVBench Workloads

KVBench is a recent key-value benchmarking suite designed to capture a range of data access patterns observed in production systems [37]. It defines five distinct workload profiles (KVBench-I – KVBench-V), each representing a unique combination of operations, key distributions, and data sizes. Tectonic can reproduce these workloads using declarative JSON specifications, enabling users to experiment with the same access behaviors under a unified and extensible framework. To run a specific workload using Tectonic, users can provide the corresponding configuration file, e.g., 1.spec.json for KVBench-I: `tectonic-cli generate -w kvbench-specs/i.spec.json`

3.3 The Tectonic Benchmark

We define seven benchmark workloads to show Tectonic's flexibility in simulating complex key-value access patterns, as shown in Table 2.

Tec-1 *(Multi-phased dynamic)* models three sequential workload phases, transitioning from a write-heavy phase to a read-heavy one and ending with a balanced mix of operations, including range queries and updates, each following a different access distribution.

Tec-2 *(Interleaved multi-distributional)* generates a workload with interleaved operations, including insert, point, and range queries, point and range deletes, and empty point queries. Each operation uses its distinct statistical distribution, ideas for stress-testing systems under mixed and concurrent loads.

Tec-3 *(Near-sorted ingestion)* simulates scenarios where data arrives mostly in order, such as time-series databases, but includes controlled disorder in key insertion order to evaluate tolerance to near-sortedness.

Tec-4 *(Soft phase transitioning)* captures a gradual transition of the workload characteristic over time, such as reducing insert rates and increasing both point and range query volume across phases.

Tec-5 *(Discrete phase transitioning)* represents a sharp transition from write-heavy to read-heavy behavior across two phases, helpful in evaluating system responsiveness under sudden access pattern changes.

Tec-6 *(Variable key-value size)* explores workloads with varying key and value sizes—ranging from 32B to 256B to simulate document stores or blob systems where entry size variability influences I/O and memory usage.

Tec-7 *(Skewed access)* focuses on skewed data access, where a small portion of keys dominate both inserts and point queries, a typical pattern in caching workloads and real-time analytics.

Together, these workloads highlight Tectonic's ability to capture complex temporal, structural, and statistical variations in realistic system benchmarks.

Table 2: Benchmark workloads supported by Tectonic

workload composition (%)	distribution of operations
Tec-1: Multi-phased dynamic	
p1: 90% insert, 10% point query; p2: 90% point query, 10% insert; p3: 20% insert, 50% range query (sel=0.01), 30% point query	p1: Zipfian (both ops.); p2: uniform (both ops.); p3: normal (all ops.)
Tec-2: Interleaved multi-distributional	
50% insert, 20% read, 10% point delete, 10% empty point query, 10% range delete (sel=0.1)	Zipfian, latest normal, uniform Pareto
Tec-3: Near-sorted ingestion	
100% insert with K=1% and L=10%	uniform
Tec-4: Slow phase transitioning	
p1: 70% insert, 20% point query, 10% range query; p2: 50% insert, 30% point query, 20% range query; p3: 30% insert, 40% point query, 30% range query	varied selectivity between 0.1 and 0.2 in p1, p2 & p3
Tec-5: Discrete phase transitioning	
p1: 40% insert, 50% point query, 10% range query; p2: 10% insert, 70% point query, 20% range query;	transition from p1 to p2 write-heavy to read-heavy
Tec-6: Variable entry size	
50% insert, 50% point query	key varies from 32B–256B value varies from 32B–256B
Tec-7: Skewed access	
50% insert (10% hot keys with a given prefix), 50% point query (90% queries with a given prefix)	10% hot keys in inserts 90% point query on hot keys

4 Experimental Evaluation

In this section, we first evaluate the performance of Tectonic against YCSB and KVBench across the six YCSB workloads (YCSB-A - YCSB-F). We use three system-level metrics: (i) end-to-end workload generation latency, (ii) operational throughput, and (iii) the peak main memory footprint, measured by the maximum resident set size. Next, we evaluate Tectonic's against KVBench using the five KVBench benchmark workloads. Finally, we show that Tectonic scales efficiently with increasing workload sizes.

Tectonic Dominates the State of the Art when Generating YCSB Workloads. To run these experiments, we set up all the benchmarks to generate an equal number of operations with the exact proportions and wrote them to an output file. The YCSB benchmark generates operations on the fly to improve memory utilization. To ensure an apples-to-apples comparison, we adapted YCSB to write all operations in an output file (using Java's `BufferedWriter`). Other benchmarks, such as KVBench and Tectonic, follow the same pattern and construct a workload file beforehand. We do not compare against RocksDB's db_bench benchmark, as it is closely tied to RocksDB internals, and collecting all operations in an output file was not straightforward. The Tectonic generates all different YCSB workloads 12× and 10× faster than KVBench and YCSB itself, respectively (Fig. 3 (A). It shows an improved throughput of ≈ 90%, Fig. 3 (B), as it performs random key generation efficiently (using the `rand_xoshiro` Rust library) while optimizing the operation generation process using `ActiveKeySet` containers and custom data structures. Tectonic does an

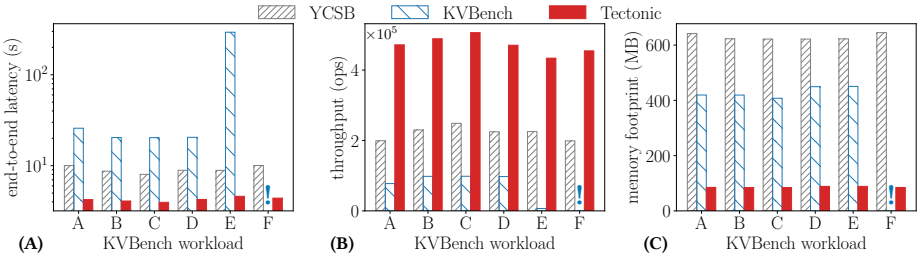

Fig. 3: Tectonic is faster than all state-of-the-art key-value generators (A) and provides better throughput (B) while using less memory (C) for all YCSB types of workloads. Note that KVBench can not simulate YCSB-F workload.

excellent job in optimizing the total memory usage, showing an improvement of 84%, Fig. 3 (C), making it capable of generating larger workloads. KVBench is unable to simulate YCSB-F, as it does not support read-modify-write, and hence, is omitted in Fig. 3. For YCSB-E, which involves short-range queries that read up to 100 records, KVBench shows higher end-to-end latency. This is because the vector is sorted every time a new range query is generated; as an insert operation appends a new key to the vector, it becomes unsorted. On the other hand, Tectonic is capable of generating range queries in two formats: (i) the YCSB format, which features an open range query with a specified *scan length*, and (ii) the KVBench format, which generates both start and end keys. For Fig. 3, Tectonic follows the YCSB way of generating range queries, which does not require sorting.

Tectonic Outperforms KVBench Across the Board. We again configure both benchmarks to produce the same number of operations with the same key and value sizes, as well as the same distribution or selectivity, if applicable. To run KVBench workloads I, III, and IV, we required preloading of inserts. In the first phase, we generate 1M insert operations, and then use this count to compute other operation compositions. In Fig. 4, we show the performance of Tectonic when generating KVBench workloads. The observations are as follows: (i) Tectonic simulated all five different KVBench workloads with 1.5 order of magnitude faster than KVBench, Fig. 4 (A); (ii) it provided more than 1.5 order of magnitude better throughput, Fig. 4 (B); and (iii) it only utilized 70% less memory than KVBench Fig. 4 (C). The workload IV in KVBench generates 50% range deletes with selectivity 0.000001 and exhibits poor throughput comparatively. Tectonic can emulate KVBench workloads more efficiently than KVBench itself. Tectonic outperforms KVBench because it uses efficient data structures to store the `ActiveKeySet`, such as Bloom filters, which enables it to achieve higher throughput with a lower memory footprint.

Tectonic Scales Better than the State of the Art. For scalability experiments, we vary the total number of operations (from 1M to 100M) along the x-axis and scale the proportions of each operation accordingly. The benchmark first generates 50% inserts for preloading, followed by 40% inserts, 5% point queries, and 5% updates in an interleaved fashion. Fig. 5(A) shows that Tec-

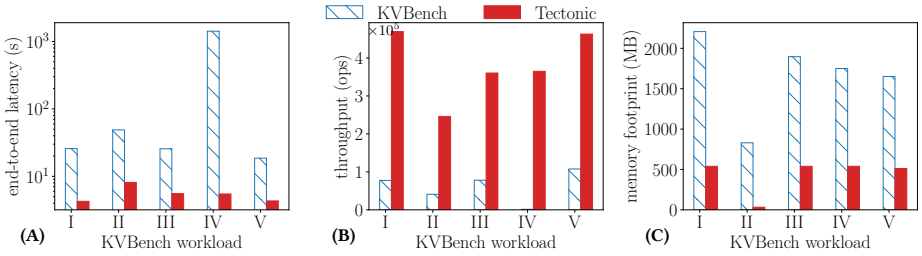

Fig. 4: Tectonic emulates all KVBench generic workloads more efficiently, showing (A) more than 1.5× improvement in end-to-end execution latency, (B) enhanced throughput, and (C) a reduced memory footprint.

tonic offers 56% and 6.4× higher throughput than YCSB and KVBench, respectively. As we increase the size of the workload, we observe that Tectonic's advantages over the state of the art hold in terms of operational throughput. This is because the workload is CPU-bound on generating random strings (as shown in Fig. 5 (A)) and Tectonic's String generation implementation is faster and efficient compared to both YCSB and KVBench. In Fig. 5 (B), however, we observe that as the workload size increases, Tectonic has a slightly higher memory footprint than YCSB. This is because workload generation in YCSB is *memoryless*, and hence, YCSB does not support operations such as empty point queries and empty point deletes. For 1M operation Tectonic still shows less memory footprint than YCSB, see Fig. 3 (C). While KVBench supports all core key-value operations, it requires significantly more metadata to enable them, requiring up to 2 orders of magnitude more memory than Tectonic.

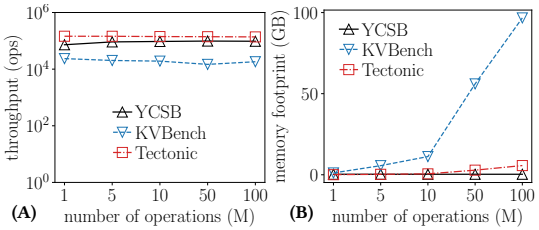

Fig. 5: Tectonic scales better than the state of the art in terms of performance at the cost of a slight increase in the memory footprint.

5 Related Work

YCSB. The Yahoo! Cloud Serving Benchmark (YCSB) [11] is a widely used key-value benchmark for evaluating the performance of cloud-native NoSQL databases. It provides a flexible architecture with support for various database backends and is easily extensible. YCSB's primary workload module, `CoreWorkload`, includes support for inserts, updates, read-modify-writes, point queries, and range queries (scans). Point queries and updates can be configured to follow distributions such as uniform, exponential, sequential, Zipfian, latest, and hotspot. However, YCSB lacks support for point and range deletes, as well as empty

point queries. Additionally, it requires point queries and updates to share the same distribution. YCSB also does not support custom key or value formats.

db_bench. db_bench [14] is a benchmarking utility tightly integrated with RocksDB [15] and LevelDB [18], offering detailed insights into internal behaviors and metrics. db_bench supports inserts, updates, read-modify-writes, point queries, range queries, point deletes, and range deletes. It can simulate empty point queries by generating keys that are guaranteed not to exist. However, modifying db_bench to serve as a general-purpose key-value benchmark requires a significant amount of development, which has led to its limited adoption.

KVBench. KVBench [37] supports a comprehensive set of operations, including inserts, updates, range queries, range deletes, as well as both empty and non-empty point queries and deletes. Each operation can be assigned its own access distribution, such as uniform, normal, beta, or Zipfian. KVBench allows limited control over operation sequencing (e.g., interleaved vs. sequential), but it does not support dynamic or time-varying workloads. Additionally, it lacks support for read-modify-writes, variable-length keys and values, and complex key formats.

BoDS. The Benchmark on Data Sortedness (BoDS) [28] is a workload suite designed to evaluate key-value and relational systems under varying degrees of data sortedness. It only supports five workload types: bulk loading, individual inserts, and mixed read/write workloads with or without preloading. BoDS allows control over data ordering using sortedness parameters K, L, and the beta distribution $B(a, b)$. However, BoDS does not support a number of basic key-value operations, such as read-modify-writes, deletes, and empty point lookups, lacks support for dynamically shifting workloads, and can not emulate complex multi-distributional workloads.

Emulating Facebook's Production Workload Traces. Cao *et al.* presented a detailed characterization of the real-world workload at RocksDB for use cases at Facebook [5]. They presented three main workloads: UDB, a MySQL-compatible storage layer for social graphs; ZippyDB, a distributed KV store for metadata; and UP2X, a distributed KV store for AI/ML services with frequent updates. They emulated the real-world workloads with YCSB and db_bench, and while db_bench was able to more accurately emulate the workloads, there is still room for improvement.

6 Conclusion

In this paper, we present Tectonic, a highly configurable and resource-efficient key-value benchmark that can accurately model complex real-world key-value workloads as well as generate dynamic workloads that change over time. Tectonic outperforms state-of-the-art key-value benchmarks, namely, YCSB and KVBench, in terms of performance and resource utilization, while scaling similarly to the state of the art. Tectonic also offers a fine-grained control over data access patterns, sophisticated key and value generation strategies, dynamic workload parameters, and support for shifting, sorted, and near-sorted workloads.

References

1. Apache: Cassandra. http://cassandra.apache.org (2023)
2. Apache: HBase. http://hbase.apache.org/ (2023)
3. Apple: FoundationDB. https://github.com/apple/foundationdb (2018)
4. Athanassoulis, M., Ailamaki, A.: BF-Tree: Approximate Tree Indexing. Proceedings of the VLDB Endowment **7**(14), 1881–1892 (2014), http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf
5. Cao, Z., Dong, S., Vemuri, S., Du, D.H.C.: Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST). pp. 209–223 (2020)
6. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J.J., Hunter, J., Barnett, M.: FASTER: A Concurrent Key-Value Store with In-Place Updates. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 275–290 (2018). https://doi.org/10.1145/3183713.3196898, http://doi.acm.org/10.1145/3183713.3196898
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) **26**(2), 4:1–4:26 (2008). https://doi.org/10.1145/1365815.1365816, https://doi.org/10.1145/1365815.1365816
8. Chatterjee, S., Jagadeesan, M., Qin, W., Idreos, S.: Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. Proceedings of the VLDB Endowment **15**(1), 112–126 (2021). https://doi.org/10.14778/3485450.3485461
9. Cheng, A., Shi, X., Kabcenell, A.N., Lawande, S., Qadeer, H., Chan, J., Tin, H., Zhao, R., Bailis, P., Balakrishnan, M., Bronson, N., Crooks, N., Stoica, I.: TAOBench: An End-to-End Benchmark for Social Networking Workloads. Proceedings of the VLDB Endowment **15**(9), 1965–1977 (2022). https://doi.org/10.14778/3538598.3538616, https://www.vldb.org/pvldb/vol15/p1965-cheng.pdf
10. CockroachDB: CockroachDB. https://github.com/cockroachdb/cockroach (2021)
11. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC). pp. 143–154 (2010). https://doi.org/10.1145/1807128.1807152, http://doi.acm.org/10.1145/1807128.1807152
12. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. ACM SIGOPS Operating Systems Review **41**(6), 205–220 (2007). https://doi.org/10.1145/1323293.1294281, http://dl.acm.org/citation.cfm?id=1323293.1294281
13. Facebook: MyRocks. http://myrocks.io/ (2023)
14. Facebook: db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools (2024)
15. Facebook: RocksDB. https://github.com/facebook/rocksdb (2024)
16. Foundation, F.S.: Gnu operating system. https://www.gnu.org/software/time/
17. Geffner, S., Agrawal, D., El Abbadi, A., Smith, T.R.: Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). pp. 328–335 (1999). https://doi.org/10.1109/ICDE.1999.754948, http://dx.doi.org/10.1109/ICDE.1999.754948

18. Google: LevelDB. https://github.com/google/leveldb/ (2021)
19. Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., Wei, W., Liu, C., Zhang, J., Li, J., Wu, X., Song, L., Sun, R., Yu, S., Zhao, L., Cameron, N., Pei, L., Tang, X.: TiDB: A Raft-based HTAP Database. Proceedings of the VLDB Endowment **13**(12), 3072–3084 (2020). https://doi.org/10.14778/3415478.3415535, http://www.vldb.org/pvldb/vol13/p3072-huang.pdf
20. Huang, G., Cheng, X., Wang, J., Wang, Y., He, D., Zhang, T., Li, F., Wang, S., Cao, W., Li, Q.: X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 651–665 (2019). https://doi.org/10.1145/3299869.3314041
21. Huynh, A., Chaudhari, H.A., Terzi, E., Athanassoulis, M.: Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. CoRR **2110.13801** (2021), https://arxiv.org/abs/2110.13801
22. Huynh, A., Chaudhari, H.A., Terzi, E., Athanassoulis, M.: Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. Proceedings of the VLDB Endowment **15**(8), 1605–1618 (2022). https://doi.org/0.14778/3529337.3529345, https://doi.org/10.14778/3529337.3529345
23. Kryczka, A.: Compaction Styles. https://github.com/facebook/rocksdb/blob/gh-pages-old/talks/2020-07-17-Brownbag-Compactions.pdf (2020)
24. Mo, D., Chen, F., Luo, S., Shan, C.: Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. CoRR **abs/2308.0** (2023). https://doi.org/10.48550/ARXIV.2308.07013, https://doi.org/10.48550/arXiv.2308.07013
25. Nuessle, C., Kennedy, O., Ziarek, L.: Benchmarking Pocket-Scale Databases. In: Performance Evaluation and Benchmarking for the Era of Cloud(s) - 11th TPC Technology Conference, TPCTC 2019, Los Angeles, CA, USA, August 26, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12257, pp. 99–115 (2019). https://doi.org/10.1007/978-3-030-55024-0{_}7, https://doi.org/10.1007/978-3-030-55024-0_7
26. Power, C., Patel, H., Jindal, A., Leeka, J., Jenkins, B., Rys, M., Triou, E., Zhu, D., Katahanas, L., Talapady, C.B., Rowe, J., Zhang, F., Draves, R., Santa, I., Kumar, A.: The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. Proceedings of the VLDB Endowment **14**(12), 3148–3161 (2021). https://doi.org/10.14778/3476311.3476390, http://www.vldb.org/pvldb/vol14/p3148-jindal.pdf
27. Raman, A., Huynh, A., Lu, J., Athanassoulis, M.: Benchmarking Learned and LSM Indexes for Data Sortedness. In: Proceedings of the International Workshop on Testing Database Systems (DBTest). p. 16–22. DBTest '24 (2024). https://doi.org/10.1145/3662165.3662764, https://doi.org/10.1145/3662165.3662764
28. Raman, A., Karatsenidis, K., Sarkar, S., Olma, M., Athanassoulis, M.: BoDS: A Benchmark on Data Sortedness. In: Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC). pp. 17–32 (2022)
29. Raman, A., Karatsenidis, K., Xie, S., Olma, M., Sarkar, S., Athanassoulis, M.: QuIT your B+-tree for the Quick Insertion Tree. In: Proceedings of the International Conference on Extending Database Technology (EDBT). pp. 451–463 (2025). https://doi.org/10.48786/EDBT.2025.36, https://doi.org/10.48786/edbt.2025.36

30. Raman, A., Sarkar, S., Olma, M., Athanassoulis, M.: Indexing for Near-Sorted Data. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). pp. 1475–1488 (2023)
31. RocksDB: Prefix seek. https://github.com/facebook/rocksdb/wiki/Prefix-Seek
32. RocksDB: Prefix Bloom Filter. https://github.com/facebook/rocksdb/wiki/Prefix-Seek#configure-prefix-bloom-filter (2020)
33. Sarkar, S., Papon, T.I., Staratzis, D., Athanassoulis, M.: Lethe: A Tunable Delete-Aware LSM Engine. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 893–908 (2020). https://doi.org/10.1145/3318464.3389757
34. Sarkar, S., Papon, T.I., Staratzis, D., Zhu, Z., Athanassoulis, M.: Enabling Timely and Persistent Deletion in LSM-Engines. ACM Transactions on Database Systems (TODS) **48**(3), 8:1–8:40 (2023). https://doi.org/10.1145/3599724, https://doi.org/10.1145/3599724
35. Sarkar, S., Staratzis, D., Zhu, Z., Athanassoulis, M.: Constructing and Analyzing the LSM Compaction Design Space. Proceedings of the VLDB Endowment **14**(11), 2216–2229 (2021). https://doi.org/10.14778/3476249.3476274, http://vldb.org/pvldb/vol14/p2216-sarkar.pdf
36. TPC: TPC-H benchmark. http://www.tpc.org/tpch/ (2021)
37. Zhu, Z., Saha, A., Athanassoulis, M., Sarkar, S.: KVBench: A Key-Value Benchmarking Suite. In: International Workshop on Testing Database Systems (DBTest). pp. 9–15 (2024). https://doi.org/10.1145/3662165.3662765, https://doi.org/10.1145/3662165.3662765