A New Benchmark Harness for Systematic and Robust Evaluation of Streaming State Stores

Esmail Asyabi Yuanli Wang John Liagouris Vasiliki Kalavri Azer Bestavros
{easyabi,yuanliw,liagos,vkalavri,best}@bu.edu
Boston University

Abstract

Modern stream processing systems often rely on embedded key-value stores, like RocksDB, to manage the state of long-running computations. Evaluating the performance of these stores when used for streaming workloads is cumbersome as it requires the configuration and deployment of a stream processing system that integrates the respective store, and the execution of representative queries to collect measurements.

To address this issue, in this paper, we start with an empirical characterization of streaming state access workloads collected from Apache Flink and RocksDB, using three publicly available datasets, and we show that the characteristics of real traces cannot be approximated with existing benchmarks. Next, we present *Gadget*, a new benchmark harness that generates realistic streaming state access workloads to enable easy and thorough performance evaluation of standalone KV stores through accurate simulation of streaming operator logic. Finally, we use *Gadget* to investigate the suitability of RocksDB as the *de facto* kv store for stream processing systems. Interestingly, we find that, although RocksDB provides robust results, it is outperformed by FASTER and BerkeleyDB in six out of eleven workloads. Our results reveal a wide performance gap between the current performance of streaming state stores and what could be achieved with workload-aware approaches.

CCS Concepts: • **Information systems** → **Stream management**; *Database performance evaluation*;

Keywords: stream processing, KV store, benchmark

ACM Reference Format:
Esmail Asyabi Yuanli Wang John Liagouris Vasiliki Kalavri Azer Bestavros. 2022. A New Benchmark Harness for Systematic and Robust Evaluation of Streaming State Stores. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3492321.3519592

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519592

1 Introduction

Stream processing technology powers numerous business applications, including continuous analytics, monitoring, fraud detection, and online recommendations [26, 33]. All major cloud providers offer stream processing as a managed service [1, 2, 4, 6] and many large companies have developed in-house streaming analytics platforms [29, 34, 39, 43, 44].

Modern stream processing systems often use persistent key-value stores to manage the state of continuous queries [8, 25, 29, 44]. Despite evidence that using a state store increases application latency by as much as an order of magnitude [7, 12, 37], there are no comprehensive performance studies that contrast various streaming state management approaches. Existing streaming benchmarks [16, 30, 38, 50] do not consider state accesses, whereas KV store benchmarks, like YCSB [31], lack the necessary tuning knobs to allow for a faithful simulation of a streaming engine's interaction with the underlying state store.

To evaluate or tune the performance of a KV store for streaming workloads, users are currently left with a single laborious option: configure and deploy a stream processing system that uses the store and execute representative queries to collect measurements. Investigating alternative store designs further requires integration with a reference streaming engine. Unfortunately, replaying an input stream is not sufficient to generate a workload, as state accesses depend on the streaming operator logic. Collecting a trace requires instrumentation to capture the state access events that operators produce when processing their input.

The work presented in this paper seeks to address this state of affairs through the development of tools and benchmarks for conducting performance evaluation of streaming state stores. First, to understand the characteristics of state access workloads in streaming applications and develop methods for accurate simulation, we perform a thorough empirical characterization study. Next, we confirm empirically that existing benchmarks, like YCSB, do not generate workloads with characteristics that accurately resemble those of real streaming state access traces, and thus cannot be used for robust evaluation of streaming state stores.

To that end, we introduce *Gadget*: a benchmark harness that enables easy and systematic performance evaluation of standalone streaming state stores. *Gadget* generates representative workloads by closely simulating the state access logic of streaming operators. It achieves high accuracy by exposing a set of configurable parameters, which are unique to streaming computation, such as the arrival rate distribution, event time skew, and watermark frequency. Currently, *Gadget* provides eleven predefined workloads, supports custom operator implementation, and offers connectors to four KV stores.

Our experimental evaluation shows that *Gadget* produces state access workloads that exhibit the same temporal and spatial locality as real traces. Furthermore, we show that YCSB is not a reliable tool to determine whether a store is suited for streaming workloads. Finally, we use *Gadget*'s workloads to evaluate current practices in streaming state management and reveal opportunities for future research.

We hope *Gadget* will become a valuable tool for designers of new stream processing systems, users of existing systems who want to (i) test different configurations for embedded state stores or (ii) use an external store to benefit from decoupling compute and state, and developers of novel KV stores who might want to optimize for streaming workloads.

Paper outline and key contributions. In Section 3, we report on the first empirical characterization study of streaming state access workloads: we use three real-world publicly available data streams to collect state access traces and analyze them in terms of (i) number and type of operations, (ii) degree of amplification, (iii) temporal and spatial locality, and (iv) working set size. In Section 4, we showcase the limitations of YCSB-generated workloads. In Section 5, we present the design and implementation of *Gadget*, a new benchmark harness for systematic and robust evaluation of standalone streaming state stores. In Section 6.1, we provide eleven workloads corresponding to common streaming operators and we empirically verify *Gadget*'s accuracy. Finally, in Section 6.2 and 6.3, we integrate *Gadget* with the RocksDB, Lethe, BerkeleyDB, and FASTER KV stores, and use it to evaluate their performance for streaming state management.

Major findings. The key findings from our workload characterization and performance evaluation work are:

1. Many state access workloads are predictable and can be accurately simulated.
2. Streaming state access workloads exhibit high event and key amplification, meaning that the state store accepts a significantly higher load than the input stream arrival rate.
3. YCSB workloads can be tuned to have either high spatial or high temporal locality, but not both. Moreover, these properties are exhibited in a significantly higher degree than what is observed in real-world streaming state access traces.
4. RocksDB, the *de facto* KV store in stream processing systems, provides robust results across *Gadget* workloads, but it is outperformed by both FASTER and BerkeleyDB in six out of eleven workloads.

We will publish *Gadget* as open-source and make all traces and results of this paper publicly available.

2 Preliminaries

In this section, we provide background on stream processing and clarify basic concepts that we use throughout the paper.

2.1 Streaming dataflow concepts

In the dataflow model [14, 26], a streaming computation is represented as a logical directed graph $G = (V, E)$, where vertices in V represent **operators** and edges in E denote **data streams**. Upon deployment, the logical graph is translated to a physical execution plan, $G' = (V', E')$, which maps operators to provisioned workers, in practice, threads. We call vertices in V' **tasks** or **instances** of a logical operator in V and edges in E' physical data channels. Tasks are typically scheduled once and are long-running. Each task is assigned to exactly one worker and each worker may execute one or more tasks of the same or different operators. The assignment is system-specific; it is computed at deployment time and remains static throughout job execution, unless a reconfiguration occurs. In a **data-parallel** execution, all tasks of an operator execute an identical logic on disjoint partitions of the input stream and they communicate with tasks of upstream and downstream operators via messages.

Each event in a streaming dataflow is associated with an **event time** that corresponds to the time when the event occurred. In general, this time is different from the wall-clock time when the event arrives at the stream processor. To track progress of the computation, many stream processors use special events called watermarks [14] that are generated at the data sources. A **watermark** with event time t_w arriving at the input of an operator means that there will be no more events with event time $t \leq t_w$. In many real applications, however, events (including watermarks themselves) may arrive at an operator out-of-order. In this case, all events whose timestamp is less than or equal to the current watermark are called **late events**. An operator will consider late events within a period of allowed lateness (*e.g.*, "k units of event time after the watermark") and will discard all late events outside this period. The watermark generation frequency and the allowed lateness can both be configured by the user.

2.2 Streaming operators

Modern stream processors support the following operators:

Windows. Windows are the most prominent streaming operators. They allow computing aggregations on the most recent

events and provide continuous fresh results to their downstream applications. *Tumbling* windows split the stream into fixed-size segments of equal length. For example, a tumbling window query in a cluster monitoring application would compute *the number of jobs submitted to the cluster every 5 seconds*. Every event in the input stream belongs to a single tumbling window. *Sliding* windows define an additional slide parameter which determines how often a new window starts. If the slide is smaller than the length, then consecutive windows overlap and input events may belong to multiple windows. For example, *every 5 minutes, compute the number of jobs submitted to the cluster during the last 30 minutes*. *Session* windows group events according to periods of activity separated by periods of inactivity. Session windows have variable length and their end is detected when no event has arrived for a given time *gap*. In the cluster monitoring application, a session window can detect job stages by grouping together tasks submitted in quick succession. Regardless of its type, a window operator is *incremental* if its function is a distributive or algebraic aggregation (*e.g.*, min, average). Otherwise, we call it *holistic* (*e.g.*, median, rank).

Joins. Streaming joins are two-input operators that find matching pairs of events in their incoming streams. To make join computations practical, streaming systems typically provide window join operators to bound the state requirements and dispose events when windows expire. Two custom join operators are more expressive and flexible than window joins. *Interval* join defines a relative time interval within which an event from one stream can match events from the other stream. This join type is useful in cases when the clocks of input sources might be distributed and thus exhibit skew. *Continuous* join is useful when the stream itself encodes a validity interval or expiration timestamp – *e.g.*, a query in a location-based service that computes *the total amount of taxi fare events for a shared taxi ride before the drop-off timestamp*.

Aggregations. Continuous aggregations compute per-key rolling aggregate values (*e.g.*, sum, count, min, max) of the input events they receive. These operators are usually lightweight but their state requirements increase over time as the keyspace size of the input stream grows.

2.3 Streaming state management

Most stream processors assume a key-value schema for input events [17, 25, 27, 35, 42] and always associate state with a key that is derived from the event according to a function. Recall that a task of a data-parallel operator is executed by exactly one worker thread and processes a disjoint partition of the operator input (cf. § 2.1). This model guarantees **single-thread access isolation** to state: any state associated with a particular key is read and modified by a single worker at any point in time. Hereafter, we represent a **state access** as a tuple $a = (p, k, v, t)$, where p is an operation (*e.g.*, `get`, `put`, `delete`, *etc.*) on a key k with value v (can be null), and t is the timestamp the operation is performed.

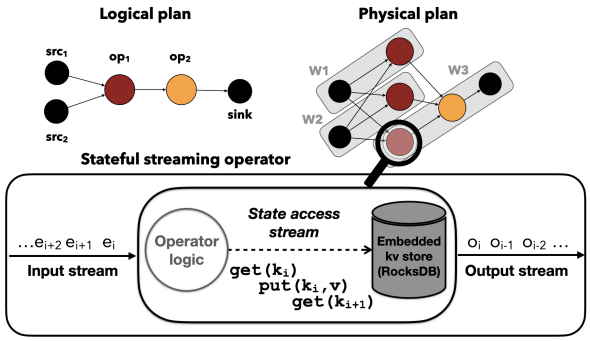


Figure 1. Streaming state access overview. A task maintains larger-than-memory state locally using its own embedded key-value store. The input stream triggers a sequence of state accesses and results in an output stream.

In this work, we consider data-parallel operators that maintain local state, possibly larger than memory, using embedded KV stores. Each operator task has its own store, as shown in Figure 1, and every incoming event e triggers a sequence of state accesses in the local store. The type and number of accesses depend on the operator logic, as we discuss in the next section. Since event processing within the same task is sequential, all requests to the state store are totally ordered and state accesses corresponding to event e_i are performed before any access due to event e_{i+1}. We define the **state access stream** as the sequence of state accesses generated by a task while processing input events. In the example of Figure 1, the state access stream is $s = (\texttt{get}, k_i, \texttt{null}, t_1), (\texttt{put}, k_i, v, t_2), (\texttt{get}, k_{i+1}, \texttt{null}, t_3), ...$, which also defines a sequence of accessed keys $(k_i, k_i, k_{i+1}, ...)$ and a sequence of operation types $(\texttt{get}, \texttt{put}, \texttt{get}, ...)$.

The embedded KV store model is rather general and is adopted by the majority of distributed stream processors, including Apache Flink, Kafka Streams, Spark Structured streaming [17] (Databricks Runtime), and Samza. External state management approaches are out of scope for this work and we discuss them in § 8.

3 Characterizing state access workloads

To accurately simulate streaming state access workloads, we first conduct a thorough empirical characterization study.

3.1 Methodology and setup

We select Apache Flink [3, 25] as a representative stream processing system with embedded state management. To capture state requests as they arrive to the KV store, we have instrumented Flink's state management layer that interacts with RocksDB [11]. Using this instrumented runtime, we collect traces of state accesses for all streaming operators of

§ 2.2. Besides `put`, `get`, and `delete`, RocksDB also supports `merge`, which is a lazy read-modify-write operation.

3.1.1 Data streams. We use three publicly available real-world data streams to drive the characterization of state access workloads. `Borg` consists of 2.5M task events and 26K job events extracted from the Google cluster usage traces [46]. `Taxi` consists of 1M taxi trips (pickup and drop-off events) and 500K corresponding taxi fare events from the 2013 NYC TLC Trip Record Data [9]. `Azure` is the full trace of 4M VM creation events of the 2017 Azure VM workload [32].

3.1.2 Configuration parameters. We evaluate all streaming operators in the *event time* domain [14] using the real timestamps provided in the input streams and punctuated watermarks with a default frequency of 100 events. Unless otherwise specified, we use 5s for window length, 1s for the window slide, and a 2min session gap. We configure interval joins with a lower bound of 2min and an upper bound of 3min. As event keys, we use the `jobID` in the `Borg` stream, the `medallionID` in `Taxi`, and the `subscriptionID` in `Azure`.

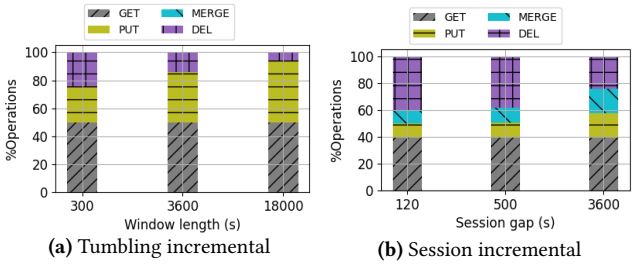
3.2 Analysis of streaming state access workloads

We start by analyzing the workload composition in terms of operation types (§ 3.2.1). The distinction between read-heavy and write-heavy workloads is an important guiding principle when evaluating KV stores. Next, we measure the degree of amplification that streaming operators cause as they transform the input stream into a state stream (§ 3.2.2). Finally, we examine state access workloads in terms of their temporal and spatial locality, and their working set size (§ 3.2.3).

3.2.1 Workload composition. Table 1 presents the percentage of reads, writes (`put` or `merge`), and deletes in the state access traces generated by different operators for all input streams (Azure is a single stream, thus, we cannot execute joins on it). We observe that the workload composition differs significantly across operators for the same input stream, and moderately across streams for the same operator.

All incremental windows, the holistic session window, joins, and aggregations generate *update heavy* workloads with an almost equal mix of `get` and `put`/`merge` operations. On the other hand, holistic tumbling and sliding window workloads are *write heavy*, having considerably higher percentages of `merge` operations compared to reads. Further, with the exception of aggregation, `delete` operations are prevalent in all workloads, ranging from 1.4% for the `Borg` continuous join to 43% for the `Taxi` holistic session window.

While this general operator behavior persists across traces, the `Taxi` stream generates a much higher percentage of deletions. In the case of windows and interval join, this difference is due to the stream's lower arrival rate. Recall that tumbling and sliding windows are configured with a 5s length by default (cf. § 3.1.2). Given such an interval, taxi rides are less frequently occurring events than job status changes and VM creation events. Similarly, a default 2min session gap is too small of an interval for taxi rides that tend to last much longer. In contrast, the number of delete operations in the continuous join workload depends on the validity period of the input events themselves (cf. § 2.2). The `Borg` stream triggers a state cleanup per job completed, while the `Taxi` stream incurs a `delete` for every passenger drop-off.



(a) Tumbling incremental **(b)** Session incremental

Figure 2. Effect of window configuration on the workload composition (`Taxi`). Smaller window lengths and session gaps produce a higher proportion of delete operations.

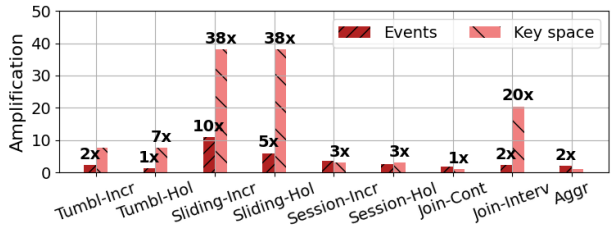


Figure 3. Event and keyspace amplification for the `Borg` stream. The state store accepts a much higher load than the stream arrival rate. All operators amplify the key space except for continuous aggregation.

Since `delete` operations occur when a window triggers, their absolute number depends on time, but their ratio over the total number of operations depends on the input rate. A stream with low input rate results in windows with few elements and respective update operations. The same effect can be observed by varying the window length or session gap for a fixed input rate. To better understand the effect of the input rate on the workload composition, we perform an experiment with the `Taxi` stream and plot the results in Figure 2. The smaller the window length (i.e., the lower the input rate) the higher the percentage of `delete` operations in the state workload, as windows contain fewer updates and expire more frequently.

Take-away: *Streaming state access workloads are update- and write-heavy. The percentage of delete operations depends on the stream's arrival rate and the window length.*

3.2.2 Amplification. To properly configure the resource allocation (*e.g.*, memtable size, cache size) of streaming state backends, developers need to estimate the request load their

	Borg				Taxi				Azure			
	GET	**PUT**	**MERGE**	**DELETE**	**GET**	**PUT**	**MERGE**	**DELETE**	**GET**	**PUT**	**MERGE**	**DELETE**
Tumbl-Incr	**0.5**	0.459	0	0.041	**0.5**	0.308	0	0.191	**0.5**	0.405	0	0.095
Sliding-Incr	**0.5**	0.459	0	0.041	**0.5**	0.308	0	0.191	**0.5**	0.405	0	0.095
Session-Incr	**0.575**	0.281	0.062	0.082	**0.399**	0.108	0.109	0.384	**0.544**	0.202	0.064	0.189
Tumbl-Hol	0.076	0	**0.847**	0.076	0.277	0	**0.446**	0.277	0.165	0	**0.669**	0.165
Sliding-Hol	0.076	0	**0.847**	0.076	0.277	0	**0.446**	0.277	0.159	0	**0.681**	0.159
Session-Hol	0.409	0	**0.477**	0.114	0.327	0	0.242	**0.430**	**0.429**	0	0.334	0.238
Join-Cont	**0.59**	0.006	0.39	0.013	**0.429**	0.281	0.143	0.147	-	-	-	-
Join-Interval	**0.446**	**0.446**	0	0.108	**0.334**	**0.334**	0	0.332	-	-	-	-
Aggregation	**0.5**	**0.5**	0	0	**0.5**	**0.5**	0	0	**0.5**	**0.5**	0	0

Table 1. Workload composition of the access traces generated by the Borg, Taxi, and Azure data streams

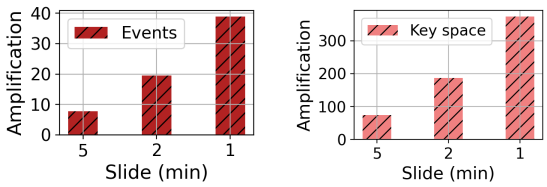


Figure 4. Effect of varying the slide of a 10-min window on event and keyspace amplification (`Taxi`). The degree of amplification is proportional to the ratio of the slide length over the window length.

applications will generate, based on the input stream characteristics. Metrics such as the stream's arrival rate and the expected number of distinct keys in the input (*e.g.*, number of concurrent cluster jobs, number of drivers) could either be known in advance or measured by monitoring the input sources. In this section, we examine how helpful such knowledge can be in estimating the state access load.

First, we find that the state store accepts a considerably higher load than the streaming operator itself. We quantify this load increase by measuring (i) *event amplification* as the number of state requests caused per event, and (ii) *key space amplification* as the ratio of distinct keys in the input stream over the number of distinct keys in the state stream. Event amplification essentially defines the request rate at the state store whereas key space amplification determines the resulting state size.

Figure 3 plots the amplification metrics for the `Borg` trace. With the exception of holistic tumbling windows, all operators generate at least 2 state accesses per input event. A stream with rate $100K$ events/s will result in $200K$ requests/s in the state store of the interval join and $400K$ requests/s in the state store of the sliding window. Keyspace amplification is significant in time-based operators, such as windows and the interval join, which use timestamps as keys to maintain state within time bounds. Continuous aggregation is the only operator that maintains the event stream properties.

In Flink, windows are mapped in state using the *W-ID* strategy [40]. Each window is represented as a KV pair, where the key is the start or end timestamp and the value is a bucket containing the window contents. When a new event arrives, the operator fetches its corresponding buckets from the KV-store, updates their contents, and writes them back. When the watermark advances, the operator identifies all expiring windows, retrieves their buckets from state, and deletes their contents. As a result, the window operator sends a pair of `get-put` (corr. `merge` for holistic windows) operations to the state for every incoming event and a pair of `get-delete` operations when windows fire. We further verify this behavior by running a sliding window experiment using the `Taxi` stream and varying the window slide. Figure 4 shows the results. For sliding windows, amplification is proportional to the ratio of the slide length over the window length, as each incoming event is assigned to $\frac{length}{slide}$ window buckets.

Finally, we examine to what degree the state access stream preserves the key distribution of the event stream. To quantify the distance between distributions, we run the KS test (Kolmogorov-Smirnov test) [23] for all operators using the `Borg` trace. Table 2 presents the results. We find that all operators distort the input distribution and none of them passes the test except continuous aggregation, which uses the input stream keys for state access.

Take-away: *The state store accepts a much higher request load than the stream arrival rate. Most workloads exhibit key distributions different from those of their respective input streams.*

3.2.3 Locality and ephemerality. We further characterize state access workloads with respect to properties that may help guide the design and configuration of caching, prefetching, and compaction components of streaming state stores. In particular, we quantify the degree of locality in state access streams and study the evolution of their working set size. For these experiments, we select three representative streaming operators: continuous aggregation as the only operator that preserves the input stream characteristics, tumbling incremental window as a commonly used time-based operator that performs incremental computation, and sliding join as a complex time-based two-input operator that performs holistic aggregation. We present the results in Figure 5 and discuss each metric in detail next.

Temporal locality indicates the likelihood that recently accessed keys will be accessed again in the near future. We

Operator	D	p-value	n	m
Tumbling-Incr	0.898	0.0	841135	1832810
Tumbling-Hol	0.896	0.0	841135	992080
Sliding-Incr	0.962	0.0	841135	9163798
Sliding-Hol	0.963	0.0	841135	4960416
Session-Incr	0.590	0.0	841135	2996940
Session-Hol	0.528	0.0	841135	2155805
Join-Cont	0.229	0.0	854582	1438628
Join-Interval	0.916	0.0	867385	1944979
Aggregation	0.0	1.0	841135	1682270

Table 2. Kolmogorov-Smirnov Test results for the `Borg` trace and the corresponding state traces. Continuous aggregation is the only operator that generates a state stream with the same distribution as the input stream.

define the temporal locality of a state access stream as the distribution of the number of unique keys accessed between consecutive operations on the same key. This definition is equivalent to the *stack distance* metric used to characterize web request workloads [15] and shown to be helpful in cache tuning [41, 52], as it can directly estimate the cache miss ratio for a given cache size. As requests come in, they are placed in a LRU stack data structure. The position of a key in the stack at the moment of its access is equal to its stack distance. Traces with small stack distances contain frequent accesses to keys which tend to be near the top of the stack.

Figure 5 (top) shows stack distance histograms of state access traces plotted in contrast to stack distance histograms of random permutations of the same trace (shuffled). Note that these are overlapping bars rather than regular stacked histograms. We observe that all three operators exhibit high temporal locality. The average stack distance computed in the state access traces is much lower than that of the shuffled traces: 53.75 as opposed to 270.40 for continuous aggregation, 1,236.95 as opposed to 9,927 for the tumbling window, and 10,627 versus 58,510 for the interval join.

Spatial locality refers to the likelihood that keys with nearby accesses in the past will also be accessed close to each other in the future. Workloads with spatial locality can benefit from prefetching mechanisms and from designs that leverage the neighborhoods of keys in the access stream to decide address space proximity. We quantify the spatial locality of a state access stream s *w.r.t.* a number $\ell \in \mathbb{N}$ by computing the distribution of the number of unique key sequences in s with maximum length ℓ.

Figure 5 (middle) plots the number of unique sequences of up to $\ell = 10$ for the state traces alongside the number of unique sequences found in the corresponding shuffled trace. The shuffled traces preserve the key popularity but destroy the sequences of accessed keys. We observe that all three operators generate workloads with high spatial locality as the total number of unique sequences observed in their traces are much lower than those found in the shuffled traces.

Working set size. As streaming computations are long-running and tend to access fresh data, we expect streaming state to be *ephemeral*. To study the evolution of streaming state, we define the *working key set* as the set of *active* keys per operator state at a specific point in time, that is the set of keys that can be accessed in the future with probability greater than zero. We further define *Time-to-Live (TTL)* as the number of time units (steps) between the first and the last access of a key in the state access stream.

To characterize state ephemerality, we sample the working set size in fixed steps of 100 operations in the state access stream and plot the results in Figure 5 (bottom). The working set of continuous aggregation increases over time, as this operator maintains as many distinct keys as those appearing in the input stream. On the contrary, tumbling window removes keys from state every time a window fires. Since keys represent window boundaries, the key space is entirely refreshed periodically and more frequently for small windows. The working set size of interval join also evolves over time, as new events are added to the state and old events are removed whenever the validity interval is exceeded.

In the case of event-time operators, the evolution of the working set further depends on the watermark frequency. Recall that watermarks indicate the stream's event time *progress* [14, 48]. When a watermark with timestamp t_w arrives at an operator, it informs the system that no future event with timestamp $t \leq t_w$ will be received in the operator's input. As a result, the operator can decide that a window expiring by $t \leq t_w$ is complete. Watermarks offer a tunable trade-off between low latency and result completeness. Eager watermarks allow for early window firings and frequent state cleanup while slow watermarks provide result confidence at the expense of higher latency and longer state TTL. To study the effect of watermark frequency on the working set size, we configure the streaming source to generate watermarks with variable frequency and run an experiment with a tumbling window and the `Azure` trace. Figure 6 plots the working set size over time for frequencies of 100 events and $1K$ events. We observe that slow watermarks increase the maximum working set by up to 3×, as windows must be kept in state longer.

Take-away: *Streaming state access workloads exhibit high temporal and spatial locality. State is ephemeral and keys have low Time-to-Live.*

4 Limitations of YCSB workloads

We now investigate whether state access traces produced by streaming operators can be approximated with YCSB. Although YCSB can be configured to generate workloads with some temporal or spatial locality, we find that none of the synthetic traces are close to the real ones according to the metrics of § 3. Interestingly, this is true even for simple streaming operators, such as continuous aggregation, whose access patterns include pairs of read/update operations that one would expect to accurately simulate using the YCSB read-modify-write workload.

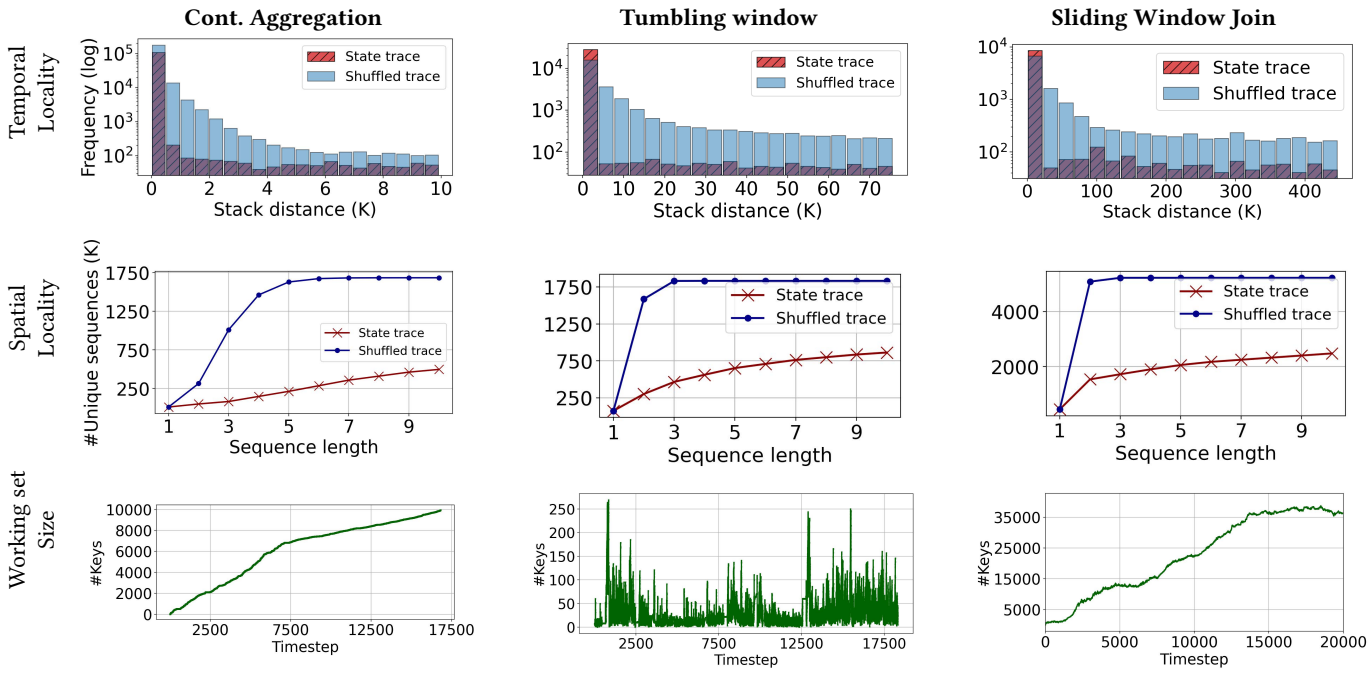


Figure 5. Locality and ephemerality characteristics of streaming state access workloads (Borg).

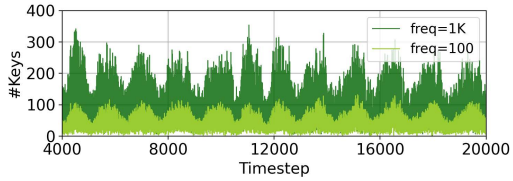


Figure 6. Effect of watermark frequency on working set size of an incremental tumbling window (Azure). Slow watermarks cause the window state to remain in the store longer, increasing the size of the working set.

Operator	p50	p90	p99.9	max
Continuous Aggr.	0.001 (1513)	206 (1630)	1675 (1651)	1676 (1652)
Tumbling Incr.	0.4 (1185)	1.4 (1633)	14.5 (1726)	21 (1727)
Sliding Join	2 (2102)	789 (4631)	2922 (5129)	2991 (5138)

Table 3. TTL (in thousand timesteps) in real traces vs TTL in the closest YCSB traces (in parentheses).

Our goal is to identify YCSB configurations that produce traces as close as possible to streaming state accesses. To do so, we generate YCSB workloads using all available request distributions (`uniform`, `zipfian`, `hotspot`, `sequential`, `exponential`, `latest`). For each YCSB workload, we set (i) the number of operations (`operationcount`), (ii) the number of distinct keys (`recordcount`), and (iii) the ratio of read/update (or read-modify-write) requests as in the respective real trace. In contrast to streaming workloads where new keys are introduced on-the-fly, YCSB assumes that distinct keys are preloaded and can be used in read/update requests as soon as the workload generation starts. New keys inserted during workload generation are not used in subsequent operations and their only purpose is to increase the size of the database. For this reason, we set the `insertproportion` parameter to zero in all synthetic workloads. We also omit delete operations, as they are not supported by YCSB.

Below we provide results for the three representative operators of § 3 (i.e., continuous aggregation, tumbling window with incremental aggregation, and sliding join) using real state access traces generated with the Borg dataset. Results for other streaming operators and datasets are similar.

Request distributions. We first compare the empirical key distributions between each real trace and all possible YCSB traces (one per built-in distribution) that have the same ratio of request types as the real trace. For each pair of traces, we map both empirical distributions to the same domain [0, `#distinct_keys`) and apply the Kolmogorov-Smirnov test. We find that the null hypothesis is rejected in all cases with significance level $\alpha = 0.001$. We also use the Wasserstein metric to quantify the distance between the real and synthetic key distributions. The Wasserstein distances range from 621 (for continuous aggregation) to 174316 (for sliding join). Our analysis shows that the built-in distributions of YCSB cannot approximate the real request distributions in the steaming workloads we consider.

Temporal and spatial locality. We further compare the real and synthetic traces with respect to their temporal and spatial locality. YCSB traces with `latest` request distribution (and in some cases `hotspot`) are the closest to the real traces in terms of temporal locality but have poor spatial locality, in most cases almost identical to that of the shuffled trace. On

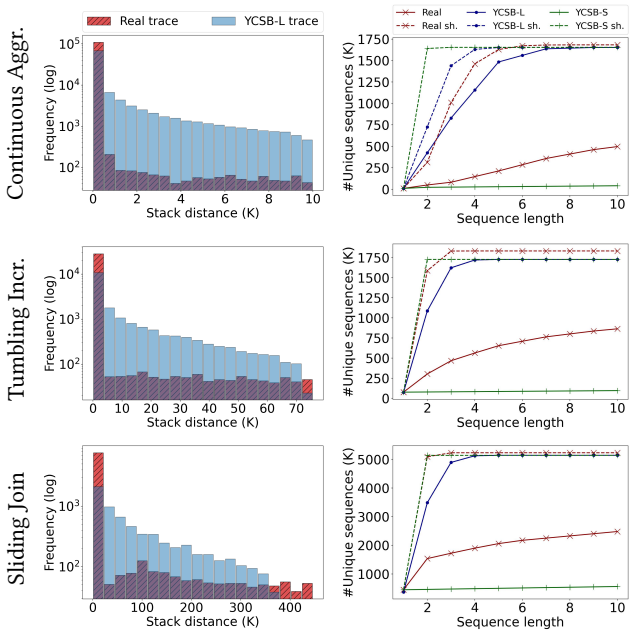


Figure 7. Stack distances for 1K random keys (left) and number of unique sequences (right) in real traces vs YCSB traces with temporal (YCSB-L) and spatial locality (YCSB-S). Dashed lines in the sequence plots correspond to shuffled traces.

the other hand, the only YCSB distribution that provides high spatial locality (but distorts temporal locality) is `sequential`. In fact, the synthetic workloads are not close to the real ones on either metric: traces with `latest` (YCSB-L) have lower temporal locality whereas traces with `sequential` (YCSB-S) have higher spatial locality compared to the real traces. Figure 7 shows the distribution of stack distances (left, overlapping bars) and the number of unique sequences (right) in the real traces and the closest YCSB traces (we also plot the number of unique sequences in the YCSB-L traces for reference). As we can see, real traces have more skewed stack distance distributions compared to YCSB-L traces (with more observations close to zero) but are closer to their shuffled counterparts in number of unique sequences compared to YCSB-S workloads. YCSB-L for continuous aggregation exhibits some spatial locality because it is generated using read-modify-write operations; the rest of the YCSB-L traces have almost the same number of unique key sequences as the respective shuffled traces.

Ephemerality. Working set sizes of YCSB workloads never decrease since YCSB does not support delete operations. Nevertheless, the synthetic traces may still exhibit some ephemerality, *e.g.*, in case certain keys are not accessed after some time. To compare the ephemerality of the real and synthetic traces, we use the distribution of TTL values. Table 3 shows different TTL percentiles for 1K randomly selected keys in the real and the closest YCSB traces. We see that the real workloads have considerably shorter TTLs compared to YCSB, especially in p50 (over 1000×) and p90 (over 5×), but also in higher percentiles for tumbling window, due to the small window length in this experiment (5*s* in event time). We also find that, in many YCSB workloads, a large percentage of keys (up to 90% in some experiments) is accessed *once*, which never happens in real streaming workloads.

5 The *Gadget* benchmark harness

We now present *Gadget*, a new benchmark harness that enables systematic evaluation of KV stores for stateful streaming applications. *Gadget* supports one or more configurable data sources and simulates the internal operations of a stream processing system to generate realistic state access workloads. *Gadget* operates in two modes: *online* and *offline*. When operating online, *Gadget* generates and issues state access requests to the KV store on-the-fly, while collecting performance measurements on latency and throughput. In offline mode, *Gadget* generates and stores a state access stream that can be replayed on demand using a built-in trace replayer. *Gadget* currently supports four KV stores with different design and performance characteristics: RocksDB [11], Lethe [47], FASTER [28], and BerkeleyDB [5].

We implemented *Gadget* in C++ in 18*K* LOC. Figure 8 shows an overview of its architecture, which consists of four core components: (i) the *event generator* that generates streams of events according to a set of user-defined properties (*e.g.*, key distribution, arrival rate, watermark frequency, *etc.*), (ii) the *driver* that simulates the internal operations of various streaming operators (*e.g.*, windows, joins, aggregations) and drives the trace generation process, (iii) the *workload generator* that produces the actual state access streams, and (iv) the *performance evaluator* that uses the generated workloads to evaluate KV store performance. Next, we describe each component in detail.

5.1 Event generation

The event generator enables *Gadget* users to configure various characteristics of the input stream, such as the arrival rate and the distribution of event keys, values and their sizes. In the example of Figure 8, event timestamps follow a Poisson process (exponential) whereas event keys follow a Zipfian distribution and the value size is constant (10 bytes).

Gadget keeps track of the input stream progress using watermarks whose frequency can also be specified by the user. To accurately simulate out-of-order events, *Gadget* exposes parameters to specify the percentage of such events and the allowed lateness period. In the example of Figure 8, the generator is configured to send one watermark every 3 time units and 2% of the generated events will appear with a (uniformly distributed) lateness of at most 3 time units from their actual event time. In practice, *Gadget* maintains an index of late events that is updated at every time step.

One salient feature of *Gadget* is that it decouples the event generator from the actual workload generator by assigning

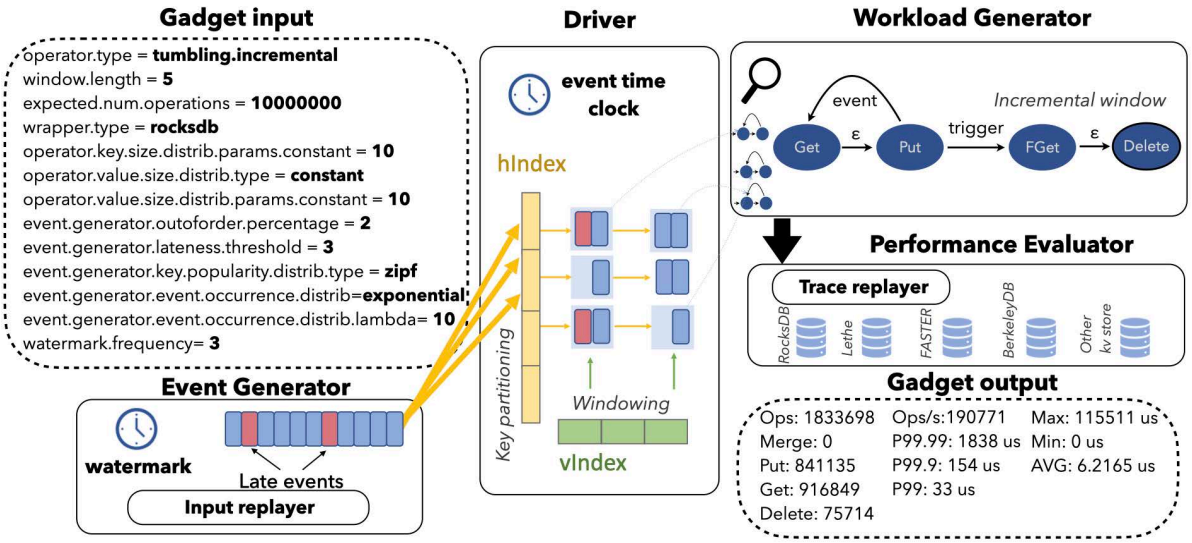


Figure 8. *Gadget* architecture overview. The event generator generates event streams according to a configuration file. *Gadget*'s driver partitions the input stream horizontally and/or vertically, depending on the logic of the specified operator. Each partition is assigned to a state machine. *Gadget*'s workload generator executes the state machines to generate the state access stream. The latter is forwarded to the performance evaluator that sends requests to the specified KV store and collects measurements.

Algorithm 1: *Gadget* driver logic

```
1  Function driver():
       // Pull and process the next batch of events
2      batch = getNext();
3      for event e in batch do
4          stateMachines = assignStateMachines(e);
5          for m in stateMachines do
6              m.run();
7          end
8      end
9  Function onWatermark():
10     stateMachines = collectExpiredStateMachines();
11     for m in stateMachines do
12         m.terminate();
13     end
```

64-bit timestamps to events, which can then be replayed using different time units. This allows *Gadget* to generate highly dense event (and subsequent state access) streams with a single thread. Besides built-in distributions (*e.g.*, zipfian, exponential, uniform, *etc.*), the event generator can also work with empirical cumulative distribution functions (ECDFs) provided by the user or even with an existing event trace like those we used in § 3. In the latter case, the event stream is replayed using the input replayer of Figure 8.

5.2 Driver

The driver's core task is to map input events to state objects and generate other necessary metadata. It maintains an index (`hIndex`) that maps event keys to state keys and, for window-based operators, it uses an additional index (`vIndex`) that maps window expiration times to state keys. In general, the event-to-state key mappings differ across operators and depend on the actual implementation. *Gadget*'s default implementation follows that of built-in operators in Apache Flink, however, users can provide alternative implementations as we describe in § 5.4.

Windowing is implemented with the *W-ID* strategy [40]. When an event arrives, the driver uses an assigner function that probes the `hIndex` to identify the list of windows the event belongs to. Then, it assigns the event to the corresponding state object(s) so that the workload generator can create the necessary state access requests in a subsequent step. On trigger, the driver identifies the expired windows using the `vIndex` and instructs the workload generator accordingly.

We stress that *Gadget* maintains only the necessary metadata to drive the workload generation process. The driver does not perform any computation on values and does not issue requests to the store. More importantly, it does not generate the actual operator state. To drive a window operator, for example, it keeps track of active window ids, their expiration times, and their sizes in number of elements. This information is sufficient to generate accurate state access streams while keeping *Gadget*'s memory footprint low.

5.3 Workload generation

The workload generator is responsible for creating the state access stream. The sequence of state accesses that an input event produces depends on the operator logic. For example, an event entering a continuous aggregation operator

```
switch(state) {
  case PutState:
    generateOp(PUT);
    state = GetState;
    done = TRUE;
    break;
  case GetState:
    generateOp(GET);
    trigger ? state = DeleteState : state = PutState;
    break;
  case DeleteState:
    generateOp(DELETE);
    done = TRUE;
    break;
}
```

Figure 9. State machine for incremental tumbling window.

incurs a pair of `get`-`put` requests to retrieve and update the aggregated value associated with the event key.

Gadget models operator logic as a *finite state machine* and provides built-in implementations for various windows, joins, and aggregations. The workload generator instantiates one state machine per state key and executes it according to the inputs provided by the driver. At each step, all KV store requests triggered by an event are generated and added to a FIFO queue before the control flow moves to the next event. Recall from § 2.3 that KV store requests are represented as tuples of the form $a = (p, k, v, t)$. The request type p and the key k are given by the state machine and the event-to-state key mappings, respectively, whereas v and t are generated according to user-defined distributions.

Figure 8 shows an example state machine for incremental tumbling window (`FGet` represents the final `get` operation that retrieves the window contents upon expiration). Each state (node) in the state machine generates a KV store request and the transitions are controlled by the *Gadget* driver.

5.4 Extending *Gadget* with new streaming operators

Algorithm 1 shows the driver logic in *Gadget*. For every batch of events, the driver makes key assignments and operates the state machines. On watermark, it retrieves expiring keys from the `vIndex`, generates final KV store requests, and cleans up state. To add a new operator, *Gadget* users need to implement three methods shown in Algorithm 1: (i) `assingState Machines()`, which generates the necessary mappings of event keys to state keys, `run()`, which defines the state machine transitions and request generation, and `terminate()`, which "closes" a state machine and cleans up state. Implementing `run()` is as simple as defining the state transitions in a `switch` statement, like the one shown in Figure 9 for the incremental tumbling window. All three methods can contain arbitrarily complex logic and have access to `hIndex`, `vIndex`, and latest seen watermark.

Based on our experience implementing Flink's operators with the *Gadget* API, we believe extending *Gadget* with a new state machine is considerably easier than adding instrumentation to a stream processing system. Recording traces from a stream processing system requires significant development effort and the expertise to identify all classes and interfaces in the source code where operators communicate with the state store. Further, collecting the traces requires configuring and deploying the full system, possibly multiple times, for each input source and operator of interest. Instrumenting the KV store could be another option, though information about the event keys and their relationship to state keys is not available at that layer. On the contrary, *Gadget* can generate traces in a lightweight manner on the user's laptop. The state machines we implemented for most operators were written in 30 lines of code or less.

5.5 State store performance evaluation

The performance evaluator uses the state access stream to assess the performance of KV stores in terms of latency and throughput. The evaluator includes a built-in trace replayer that replays the state access stream and sends the respective requests to the underlying store. Besides *Gadget*-generated traces, the replayer can also replay workloads generated with other benchmarks, such as YCSB, and can be configured with a *service rate* to speed up or slow down the trace arbitrarily.

By default, state access streams in *Gadget* include four types of operations $p = \{\texttt{get}, \texttt{put}, \texttt{merge}, \texttt{delete}\}$, which correspond to the operations supported by RocksDB (and Lethe). Other KV stores have a different set of operations, for example, BerkeleyDB and FASTER do not support `merge` requests and they instead have implementations for in-place updates (respectively `update` and `rmw`). The performance evaluator is responsible for translating the requests in the state access stream to the requests supported by the underlying KV store. To add a new KV store to *Gadget*, a user needs to implement a C++ wrapper that maps `get`, `put`, `merge`, `delete` operations to the corresponding operations in the KV store. For example, the RocksDB API has direct calls for all *Gadget* operations, while FASTER maps `get` to `read`, `put` to `upsert`, and `merge` to `rmw`.

6 Evaluation

Our evaluation is structured into three parts. In § 6.1, we empirically show that *Gadget* can generate realistic streaming state access workloads that exhibit the same temporal and spatial characteristics as those observed in real traces. In § 6.2, we examine how the trace characteristics can impact the measured KV store performance. We demonstrate that when using *Gadget* workloads for evaluation, the throughput and latency results are close to those obtained with real traces. In § 6.3, we use *Gadget* to evaluate the performance of four KV stores for eleven streaming workloads. Finally, in § 6.4 we evaluate RocksDB with concurrent operators.

Experimental setup. We run all experiments on a dual-socket machine equipped with 12-core Intel Xeon 4116 CPU running at 2.1 GHz, 32*GB* of RAM, and a 512*GB* PC400 NVMe (SK hynix). We use Ubuntu 20.04 (Linux kernel version 5.4). We configure the memory portion of the various KV stores as follows. Lethe and RocksDB have two 128*MB* write buffers (memtables) and a 64*MB* cache. We further set the Lethe delete threshold to 10*s*. We use the B$^+$Tree version of BerkeleyDB with a 256*MB* cache. The FASTER log and hash index use 256*MB* and 64*MB* respectively. We use the default values for all other configuration options. We repeat all experiments at least three times and report mean values.

6.1 How close are *Gadget* traces to real traces?

In this section, we show that *Gadget* faithfully simulates streaming state accesses and can produce workloads that exhibit the characteristics of real traces. We configure *Gadget* to generate workloads for the three representative operators of § 3.2.3 (continuous aggregation, tumbling incremental window, and sliding join). In these experiments, we use `Borg` as the input stream and configure *Gadget* with the same parameters we used for Flink in § 3.1.2. Next, we analyze the generated traces and compare them to the real ones in terms of temporal and spatial locality. Figure 10 plots the histogram of stack distances and the number of unique sequences.

For all operators, *Gadget* produces a trace that consists of an almost identical number of unique sequences as the real trace. The distribution of stack distances in *Gadget* traces is also very close to that of real traces. In the case of sliding join, generating the exact sequence of keys found in the real trace is challenging, due to non-deterministic source scheduling. Specifically, the order of state accesses that a join generates depends on the order the events arrive from its sources. When simulating a two-input operator, *Gadget* pulls events from each source in a round-robin fashion but in the real stream processing system source tasks might be scheduled by the OS or by custom scheduling methods.

We repeated this experiment for all operators of § 3 and the results are similar to those in Figure 10. These results indicate that *Gadget* can generate workloads that exhibit the same degree of temporal and spatial locality as real traces.

6.2 Are *Gadget* workloads valuable in practice?

While *Gadget* can closely approximate locality in real state access traces, in § 4 we empirically demonstrated that YCSB cannot. In this section, we examine how differences in trace locality affect the KV store performance in practice. We expect that a *representative* workload will produce performance results close to those achieved when using a real trace.

We use the YCSB workloads of § 4 that are manually tuned to be as close as possible to those generated by continuous aggregation, tumbling incremental window, and sliding join operators. These are the YCSB workloads with sequential,

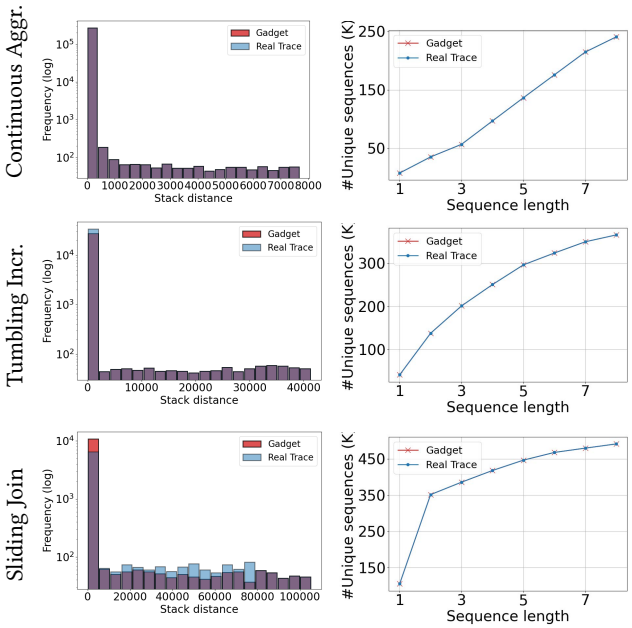


Figure 10. Stack distances (overlapping bars) and unique sequences in *Gadget* and real workloads. *Gadget*'s traces exhibit very similar spatial and temporal locality with the real traces.

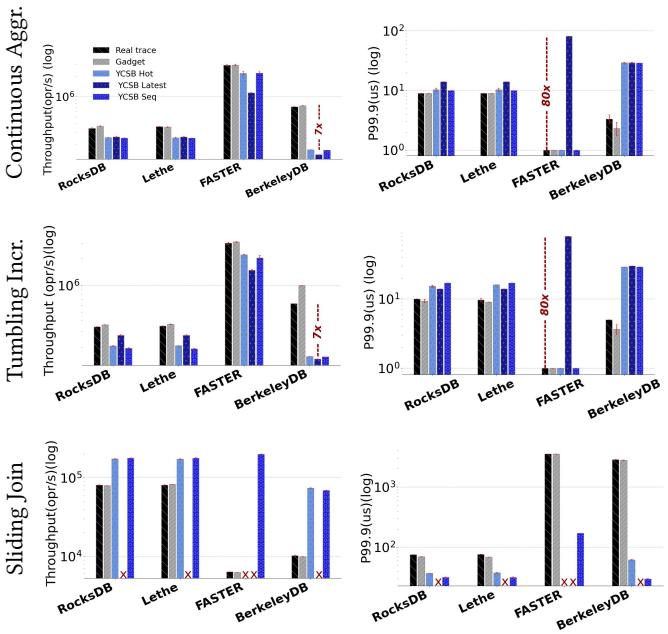


Figure 11. Throughput and latency measured with *Gadget*, YCSB, and real traces. *Gadget* results are close to those obtained with real traces. On the contrary, when using manually tuned YCSB workloads, the reported performance differs by up to an order of magnitude.

hotspot, and latest distributions. For each operator, we configure the YCSB request ratio, key/value sizes, and number of keys to be equal to those of the respective real trace. We then use the real and *Gadget* traces to drive experiments with all

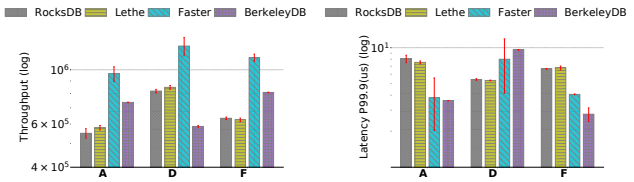


Figure 12. Performance evaluation results for all KV stores in *Gadget* using three core YCSB workloads.

four KV stores. We replay the workloads with the *Gadget* trace replayer on all KV stores and measure throughput and tail latency (p99.9) for $2M$ operations.

Figure 11 plots the results. We observe that the performance achieved with *Gadget* workloads is very close to that measured using the real traces, for all operators and KV stores. On the contrary, when using the tuned YCSB workloads, the reported throughput and latency vary significantly. For BerkeleyDB, YCSB workloads result in 7× lower throughput for the aggregation and the tumbling window operators. Further, using YCSB-latest leads to 80× higher tail latency for FASTER on continuous aggregation. In the case of sliding join, the same workload causes a major performance degradation for all KV stores. Another notable result is that, according to YCSB workloads, BerkeleyDB offers the worst throughput for the two incremental operators (continuous aggregation and tumbling window). In reality, however, BerkeleyDB outperforms both RocksDB and Lethe, which is consistent with the result we get when using *Gadget* traces.

These results demonstrate that *Gadget* is a valuable tool for accurate performance evaluation of streaming state stores.

6.3 *Gadget* in action: Evaluating streaming state stores

Finally, we demonstrate *Gadget* in action to evaluate the suitability of four KV stores for streaming state management. Without access to *Gadget*, a developer looking for a store tailored to streaming workloads might resort to YCSB, which has been used by several studies for other application scenarios [20, 28, 36, 54, 56, 57]. We adopt this approach as our baseline and use the YCSB core workloads A (50% reads, 50% writes), D (read latest), and F (read-modify-write). For this experiment, the key size is 8 bytes (the default key size of YCSB), the value size is 256 bytes, and we configure all workloads with $1K$ keys and $2M$ operations. Figure 12 shows throughput and latency results for zipfian key distribution (using uniform distribution produces comparable results). FASTER achieves higher throughput than all other KV stores across workloads but exhibits high tail latency for the read-heavy workload. RocksDB and Lethe outperform BerkeleyDB for the read-heavy workload (D), whereas BerkeleyDB has superior performance for the update-heavy workloads (A, F).

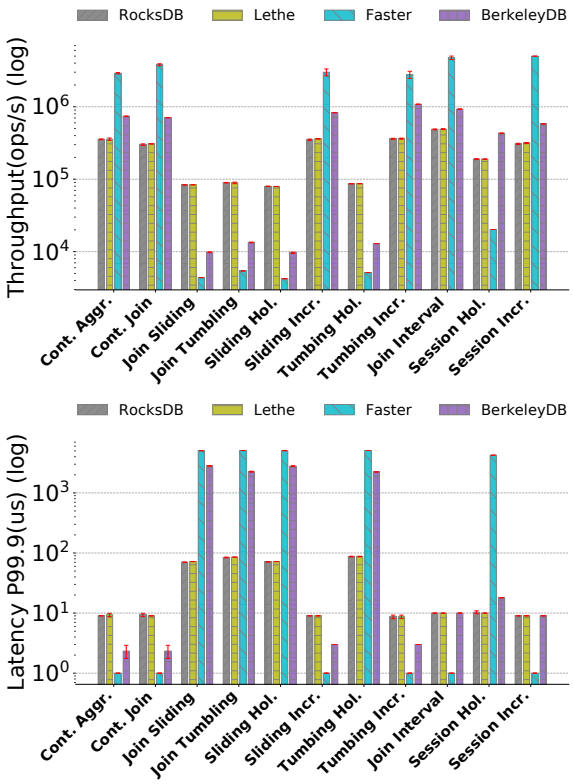


Figure 13. Performance evaluation of streaming state stores using *Gadget*. RocksDB is outperformed in six out of eleven workloads but offers robust performance for all operators.

Next, we repeat the experiment using all *Gadget* workloads. For window operators, we configure the length to $5s$, the slide to $1s$, and the session gap to $2min$. Figure 13 plots throughput and tail latency for all KV stores. We see that RocksDB, the de facto KV store in stream processing systems, is significantly outperformed in six out of eleven workloads by both FASTER and BerkeleyDB. The rest five workloads, where RocksDB and Lethe provide considerably higher throughput and lower latency than other stores, are all generated by holistic window-based operators (the only exception is the holistic session window for which BerkeleyDB achieves the best throughput). Recall that holistic window operators collect their input events into buckets and apply the aggregation function on trigger. As a result, if the KV store does not support lazy updates (such as `merge` in RocksDB), inserting an event to a window requires reading and copying a growing vector. This is the reason why FASTER and BerkeleyDB cannot achieve high throughput for holistic operators.

Overall, RocksDB and Lethe provide *robust* results: their tail latency does not exceed $100us$ for any workload and remains below $10us$ in many cases. If we can only select a single store for streaming state management, RocksDB is indeed the most sensible choice available today.

6.4 Evaluating concurrent operators

Gadget is designed according to the principles of the dataflow computation model, which guarantees a single writer task per key in the input stream. Even so, the model does not restrict the store's physical deployment and it permits multiple tasks concurrently accessing the same store. This setting can be easily evaluated by running multiple *Gadget* instances concurrently and configuring them all to access the same store instance. We perform a simple experiment to demonstrate this scenario.

We generate two workloads of $1M$ operations for (i) an incremental sliding window and (ii) a holistic sliding window operator. We configure both operators with a $5s$ window length and a $1s$ slide. Figure 14 shows operator performance when RocksDB is accessed by a single streaming operator and when it is concurrently accessed by two operators of the same (*Concurrent-A*) and different types (*Concurrent-B*). We observe that the incremental window operator has 1.7× lower throughput and 1.5× higher latency when sharing the store with another operator of the same type, while the impact is less significant when it is co-located with a holistic window. For the holistic window, co-location results in ~ 1.4× lower throughput and ~ 1.03× higher latency compared to when accessing the KV store instance in isolation.

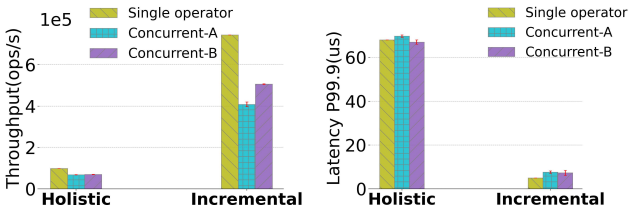


Figure 14. Performance of concurrent operators on the same RocksDB instance. *Concurrent-A* shows performance when co-locating two operators of the same type. *Concurrent-B* shows performance when co-locating two operators of different types.

6.5 Discussion

Our experimental results reveal some interesting patterns in the behavior of KV stores when used for streaming workloads. We find that LSM-trees perform well for holistic aggregates because they support lazy updates. Recall that holistic operators need to collect window contents in a data structure of variable size. RocksDB can efficiently append new values to the log and lazily collect the window contents on trigger. On the other hand, FASTER and BerkeleyDB can only support such an operation by reading, updating, and writing vectors of growing size. Hash-based and B$^+$-trees support in-place updates and are better for incremental operations. For example, FASTER outperforms RocksDB by an order of magnitude for such operators due to efficient O(1) lookups and in-place updates. These results suggest there is a wide gap between the current performance of streaming state management and what could be achieved with workload-aware approaches. Streaming systems could improve throughput and latency by an order of magnitude if they switch to hash-based or B$^+$-tree based stores for incremental operations. This is an interesting research direction for future work.

7 Related work

Workload Characterization. Even though the literature in workload characterization is rich, streaming state access traces have not been analyzed before. Past studies have considered the characteristics of web requests [15], social network services [24, 55], distributed file systems [21], VM deployments in cloud platforms [32], in-memory KV stores [19], and others. Various metrics have been used to understand and optimize performance, including request composition, KV-pair hotness distribution, key-space locality and temporal patterns, key and value sizes, working set sizes, and TTL. For example, Cao et al. [24] show that Facebook workloads exhibit high key-space locality, while Wires et al. [53] estimate miss ratio with an optimized stack distance data structure. In this work, we additionally consider metrics that are unique to streaming state store workloads, such as event and key space amplification. Further, we study how stream and operator properties, such as watermark frequency and window length, affect the corresponding metrics.

KV store benchmarks. YCSB [31] is the most widely-used benchmark for KV stores. It provides various workloads with different request ratios and key distributions. Many previous studies have shown that YCSB workloads do not exhibit the characteristics of real workloads for several application areas [22, 24, 45]. Our results are in agreement with and extend previous findings for streaming state access workloads. We also show that YCSB can be tuned to produce workloads with either spatial or temporal locality but not both and not close to the degree exhibited in streaming traces. Cao et al. [24] propose a new benchmark to generate request traces that preserve the key-space locality and temporal patterns of real traces at Facebook, but their tool is tailored around RocksDB and cannot be used to evaluate other KV stores. Pitchumani et al. [45] extend YCSB to support configurable inter-arrival times between requests, a functionality that is also provided by *Gadget*. Other benchmarks, such as LinkBench [18] and BigDataBench [51], do not consider temporal patterns and, thus, cannot generate realistic state access streams. Most importantly, none of the aforementioned tools can be used to assess the impact of the input stream characteristics (such as watermark frequency, window sizes, late events, *etc.*) to the KV store performance.

8 Discussion and future work

The results of our characterization study and our experience from building and using *Gadget* reveal many interesting opportunities for future work. *Gadget* facilitates deeper experimental analysis of streaming state stores and can enable automatic KV store configuration, evaluation of novel store designs, and optimization of stateful operators. For instance, our temporal locality analysis could be used to provide automatic cache size tuning in state stores and our spatial locality findings can guide the design of novel prefetching mechanisms. Another interesting direction is to control the frequency of compactions in LSM-based stores by leveraging the fact that delete operations in streaming workloads are highly predictable. Finally, even though we have considered embedded state in this paper, some streaming frameworks, such as MillWheel [13] and Pravega [10], rely on distributed KV stores. We believe that *Gadget* can be easily extended to support evaluation of external state management approaches [49] by running multiple concurrent instances of the workload generator and implementing the respective KV store wrappers.

Acknowledgements

We thank the anonymous EuroSys reviewers for their insightful comments and our shepherd Jean-Pierre Lozi for his guidance in improving the paper. This work was partially supported by a Google DAPA award.

References

[1] Alibaba Realtime Compute. https://www.alibabacloud.com/product/realtime-compute. Last access: October 2021.

[2] Amazon Kinesis. https://aws.amazon.com/kinesis/. Last access: October 2021.

[3] Apache Flink. https://flink.apache.org/. Last access: October 2021.

[4] Azure Stream Analytics. https://azure.microsoft.com/en-us/services/stream-analytics/. Last access: October 2021.

[5] Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html. Last access: October 2021.

[6] Google Cloud Dataflow. https://cloud.google.com/dataflow. Last access: October 2021.

[7] How to manage your RocksDB memory size in Apache Flink. https://www.ververica.com/blog/manage-rocksdb-memory-size-apache-flink. Last access: October 2021.

[8] Kafka Streams Internal Data Management. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management. Last access: October 2021.

[9] NYC TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Last access: October 2021.

[10] Pravega. https://pravega.io. Last access: October 2021.

[11] RocksDB. https://rocksdb.org/. Last access: October 2021.

[12] The RocksDB State Backend. https://ci.apache.org/projects/flink/flink-docs-release-1.9/ops/state/state_backends.html#the-rocksdbstatebackend. Last access: October 2021.

[13] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proceedings of the the VLDB Endowment*, 6(11):1033–1044, Aug. 2013.

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment*, 2015.

[15] V. Almeida, A. Bestavros, M. Crovella, and A. De Oliveira. Characterizing reference locality in the www. In *Fourth International Conference on Parallel and Distributed Information Systems*, pages 92–103. IEEE, 1996.

[16] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 480–491. Morgan Kaufmann, 2004.

[17] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613, New York, NY, USA, 2018. ACM.

[18] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1185–1196, New York, NY, USA, 2013. Association for Computing Machinery.

[19] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[20] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, Boston, MA, Feb. 2019. USENIX Association.

[21] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 198–212, 1991.

[22] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[23] L. Bol'Shev and N. Smirnov. Tables in mathematical statistics. *Tables in Mathematical Statistics [in Russian*, 1965.

[24] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, Feb. 2020. USENIX Association.

[25] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, Aug. 2017.

[26] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. Beyond analytics: the evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 2651–2658, 2020.

[27] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[28] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 275–290, New York, NY, USA, 2018. ACM.

[29] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[30] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, et al. Benchmarking streaming computation engines at yahoo! *Tech. Rep.*, 2015.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[32] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[33] M. Dayarathna and S. Perera. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, 51(2):1–36, 2018.

[34] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[35] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment*, 12(9), 2019.

[36] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, Feb. 2019. USENIX Association.

[37] V. Kalavri and J. Liagouris. In support of workload-aware streaming state management. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[38] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1507–1518. IEEE Computer Society, 2018.

[39] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[40] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322, 2005.

[41] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[42] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[43] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1591–1602. IEEE, 2020.

[44] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[45] R. Pitchumani, S. Frank, and E. L. Miller. Realistic request arrival generation in storage benchmarks. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2015.

[46] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.

[47] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 893–908, New York, NY, USA, 2020. Association for Computing Machinery.

[48] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274, 2004.

[49] Q.-C. To, J. Soto, and V. Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27(6):847–872, Dec. 2018.

[50] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark—A Benchmark for Queries over Data Streams. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[51] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[52] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 335–349, 2014.

[53] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, Oct. 2014. USENIX Association.

[54] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[55] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, Nov. 2020.

[56] S. Zheng, M. Hoseinzadeh, and S. Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, Boston, MA, Feb. 2019. USENIX Association.

[57] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, page 3, USA, 2012. USENIX Association.

A Artifact Appendix

A.1 Abstract

Gadget is a benchmark for accurate and easy evaluation of KV stores used for stream processing systems. *Gadget* digests event stream(s) and simulates the internals of a stream processing system to generate realistic state access workloads. It then issues state access requests to KV stores and collects performance measurements on latency and throughput.

A.2 Access links & Requirements

A.2.1 Access links. *Gadget* is accessible at the following GitHub link:
https://github.com/CASP-Systems-BU/Gadget or using the following DOI:
`10.5281/zenodo.6347736`
(https://zenodo.org/record/6347736).

A.2.2 Hardware dependencies. We run all our experiments on a dual-socket machine equipped with a 12-core Intel Xeon 4116 CPU running at 2.1 GHz, 32 GB of RAM, and 512GB PC400 NVMe (SK Hynix) hard disk.

A.2.3 Software dependencies. We ran the experiments on Ubuntu 20.04 (Linux kernel version 5.4). To evaluate a KV store with *Gadget*, the KV store must be installed on the system. We provide a container (*Gadget* container), which hosts all KV stores evaluated in this project (RocksDB, Lethe, Faster, and BerkeleyDB).

A.3 Reproducing the Paper Results

Please take the following steps to reproduce the paper results:

1. Instantiate the following public profile on CloudLab:
   https://www.cloudlab.us/p/easyabi/gadget50
   This profile gets an m510 machine from the CloudLab Utah cluster, downloads docker, and installs all required software. Note that the CloudLab m510 machine does not match the hardware used in the evaluation of this paper, but it is close enough to reproduce the main paper claims.
2. Inside the machine, perform the following commands to download the *Gadget* container from the docker hub and run the container:
   ```
   cd /local
   sudo sh init.sh
   ```
3. Inside the container, perform the following commands to conduct the experiments:
   ```
   cd /home/gadget/build/src/
   ./runAllExprs.sh
   ```
   This command runs all experiments. Once finished, the results will be in the following folders: `firstExpr/ secondExpr/ thirdExpr/`. The experiments should take around eight hours to complete.

More detailed instructions for reproducing the paper results can be found at: https://github.com/CASP-Systems-BU/Gadget/tree/main/reproduceEuroSysResults.

A.4 Install and Run *Gadget*

Please see https://github.com/CASP-Systems-BU/Gadget for detailed instructions of installing and running *Gadget*.

A.4.1 Configure *Gadget*. Please see https://github.com/CASP-Systems-BU/Gadget/blob/main/configs for a detailed description of *Gadget* configuration.

A.4.2 Experiments with *Gadget*. Please see https://github.com/CASP-Systems-BU/Gadget/blob/main/experiments for a detailed description of performing experiments with *Gadget*.

A.4.3 *Gadget* Source Code. Please see https://github.com/CASP-Systems-BU/Gadget/blob/main/src for a detailed description of the *Gadget* source code.