

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

Anonymous Author(s)

Submission Id: 578

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 *SysX*, 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 *SysX* 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, other kv stores outperform it in six out of eleven workloads, providing up to 10 \times higher throughput.

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 *SysX*: a benchmark harness that enables easy and systematic performance evaluation of standalone streaming state stores. *SysX* 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, *SysX* provides eleven predefined workloads, supports custom operator implementation, and offers connectors to four kv stores.

Our experimental evaluation shows that *SysX* 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 *SysX*’s workloads to evaluate current practices in streaming state management and reveal opportunities for future research.

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 *SysX*, 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, which we use to empirically verify *SysX*'s accuracy. Finally, in Section 6.2 and 6.3, we integrate *SysX* 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 *SysX* workloads, but it is outperformed by both FASTER and BerkeleyDB in six out of eleven workloads.

We will publish *SysX* 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

221 input events they receive. These operators are usually light-
 222 weight but their state requirements increase over time as the
 223 keyspace size of the input stream grows.

224 225 2.3 Streaming state management

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

238 In this work, we consider data-parallel operators that
 239 maintain local state, possibly larger than memory, using
 240 embedded kv stores. Each operator task has its own store, as
 241 shown in Figure 1, and every incoming event e triggers a se-
 242 quence of state accesses in the local store. The type and num-
 243 ber of accesses depend on the operator logic, as we discuss in
 244 the next section. Since event processing within the same task
 245 is sequential, all requests to the state store are totally ordered
 246 and state accesses corresponding to event e_i are performed
 247 before any access due to event e_{i+1} . We define the **state ac-
 248 cess stream** as the sequence of state accesses generated by a
 249 task while processing input events. In the example of Figure 1,
 250 the state access stream is $s = (\text{get}, k_i, \text{null}, t_1), (\text{put}, k_i, v, t_2),$
 251 $(\text{get}, k_{i+1}, \text{null}, t_3), \dots$, which also defines a sequence of ac-
 252 cessed keys $(k_i, k_i, k_{i+1}, \dots)$ and a sequence of operation types
 253 $(\text{get}, \text{put}, \text{get}, \dots)$.

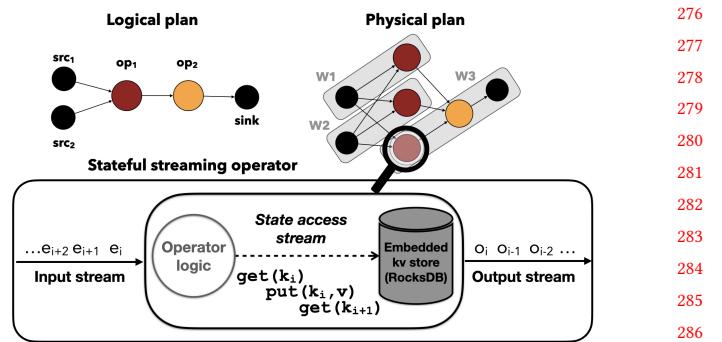
254 The embedded kv store model is rather general and is
 255 adopted by the majority of distributed stream processors,
 256 including Apache Flink, Kafka Streams, Spark Structured
 257 streaming [17] (Databricks Runtime), and Samza. External
 258 state management approaches are out of scope for this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

260 261 3 Characterizing state access workloads

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

264 265 3.1 Methodology and setup

266 We select Apache Flink [3, 25] as a representative stream
 267 processing system with embedded state management. To
 268 capture state requests as they arrive to the kv store, we have
 269 instrumented Flink’s state management layer that interacts
 270 with RocksDB [11]. Using this instrumented runtime, we
 271 collect traces of state accesses for all streaming operators of
 272 § 2.2. Besides put, get, and delete, RocksDB also supports
 273 merge, which is a lazy read-modify-write operation.



274 275 276 **Figure 1.** Streaming state access overview. A task maintains
 277 larger-than-memory state locally using its own embedded key-
 278 value store. The input stream triggers a sequence of state accesses
 279 and results in an output stream.

280 281 3.1.1 Data streams. We use three publicly available real-
 282 283 world data streams to drive the characterization of state
 284 285 access workloads. Borg consists of 2.5M task events and
 286 287 26K job events extracted from the Google cluster usage
 288 289 traces [46]. Taxi consists of 1M taxi trips (pickup and drop-
 289 290 off events) and 500K corresponding taxi fare events from the
 290 291 2013 NYC TLC Trip Record Data [9]. Azure is the full trace of
 291 292 4M VM creation events of the 2017 Azure VM workload [32].

292 293 3.1.2 Configuration parameters. We evaluate all stream-
 294 295 ing operators in the *event time* domain [14] using the real
 296 297 timestamps provided in the input streams and punctuated
 298 299 watermarks with a default frequency of 100 events. Unless
 299 300 otherwise specified, we use 5s for window length, 1s for the
 300 301 window slide, and a 2min session gap. We configure interval
 301 302 joins with a lower bound of 2min and an upper bound of
 302 303 3min. As event keys, we use the jobID in the Borg stream, the
 303 304 medallionID in Taxi, and the subscriptionID in Azure.

3.2 Analysis of streaming state access workloads

311 312 We start by analyzing the workload composition in terms of
 313 314 operation types (§ 3.2.1). The distinction between read-heavy
 314 315 and write-heavy workloads is an important guiding principle
 315 316 when evaluating kv stores. Next, we measure the degree of
 316 317 amplification that streaming operators cause as they trans-
 317 318 form the input stream into a state stream (§ 3.2.2). Finally, we
 318 319 examine state access workloads in terms of their temporal
 319 320 and spatial locality, and their working set size (§ 3.2.3).

321 322 3.2.1 Workload composition. Table 1 presents the per-
 322 323 centage of reads, writes (put or merge), and deletes in the
 323 324 state access traces generated by different operators for all
 324 325 input streams (Azure is a single stream, thus, we cannot
 325 326 execute joins on it). We observe that the workload composi-
 326 327 tion differs significantly across operators for the same input
 327 328 stream, and moderately across streams for the same operator.

328 329 All incremental windows, the holistic session window,
 329 330 joins, and aggregations generate *update heavy* workloads

	Borg				Taxi				Azure				386
	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	388
Sliding-Incr	0.5	0.459	0	0.041	0.5	0.308	0	0.191	0.5	0.405	0	0.095	389
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	390
Tumbl-Hol	0.076	0	0.847	0.076	0.277	0	0.446	0.277	0.165	0	0.669	0.165	391
Sliding-Hol	0.076	0	0.847	0.076	0.277	0	0.446	0.277	0.159	0	0.681	0.159	392
Session-Hol	0.409	0	0.477	0.114	0.327	0	0.242	0.430	0.429	0	0.334	0.238	393
Join-Cont	0.59	0.006	0.39	0.013	0.429	0.281	0.143	0.147	-	-	-	-	394
Join-Interval	0.446	0.446	0	0.108	0.334	0.334	0	0.332	-	-	-	-	395
Aggregation	0.5	0.5	0	0	0.5	0.5	0	0	0.5	0.5	0	0	396

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

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.

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

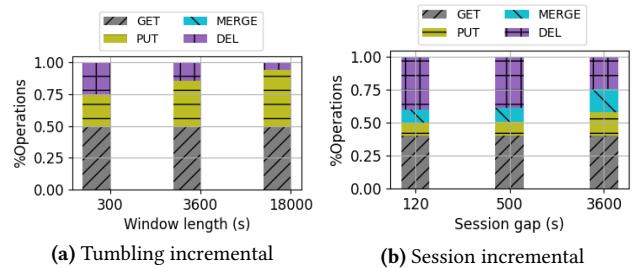


Figure 2. Effect of window configuration on the workload composition (Taxi). Smaller window lengths and session gaps produce a higher proportion of delete operations.

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 2M requests/s in the state store of the interval join and almost 4M 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

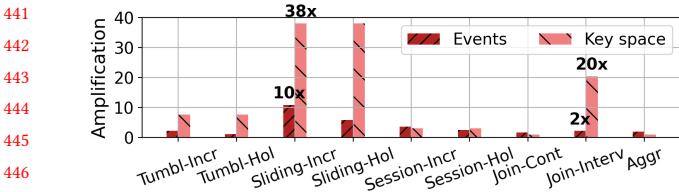


Figure 3. Event and keyspace amplification for the Borg stream. The state store accepts a much higher load than the stream arrival rate. All operators amplify the key space except for continuous aggregation.

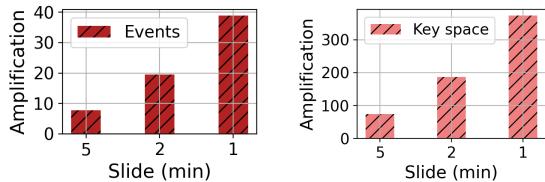


Figure 4. Effect of varying the slide of a 10-min window on event and keyspace amplification (Taxi). The degree of amplification is proportional to the ratio of the slide length over the window length.

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 $\frac{\text{length}}{\text{slide}}$ 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

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.

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 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). 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 .

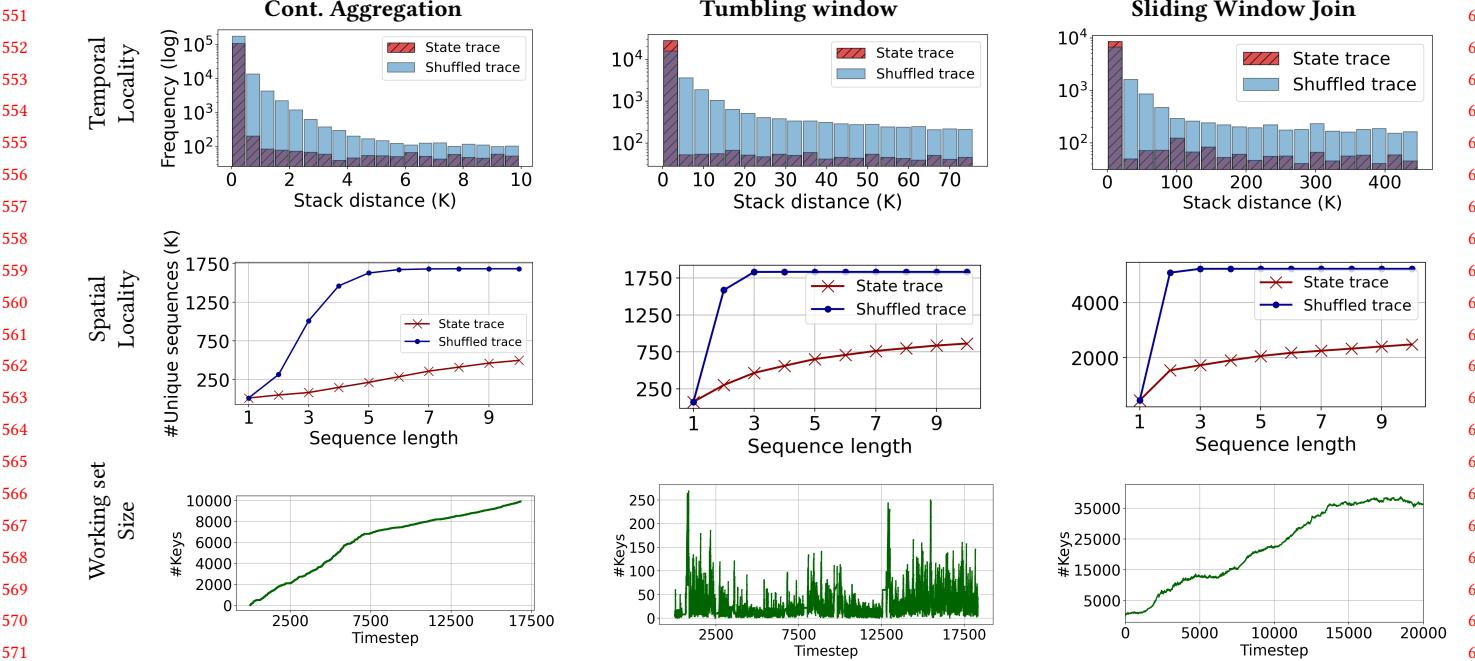


Figure 5. Locality and ephemerality characteristics of streaming state access workloads (Borg).

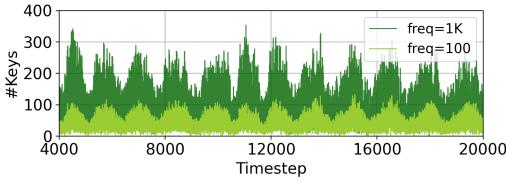


Figure 6. Effect of watermark frequency on working set size of an incremental tumbling window (Azure). Slow watermarks cause the window state to remain in the store longer, increasing the size of the working set.

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 \times , 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.

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

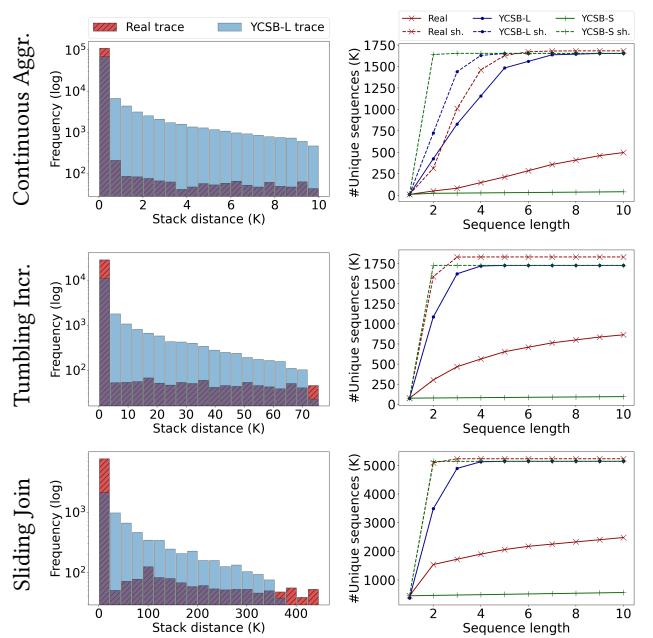


Figure 7. Stack distances for 1K random keys (left) and number of unique sequences (right) in real traces vs YCSB traces with temporal (YCSB-L) and spatial locality (YCSB-S). Dashed lines in the sequence plots correspond to shuffled traces.

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).

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 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) 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.

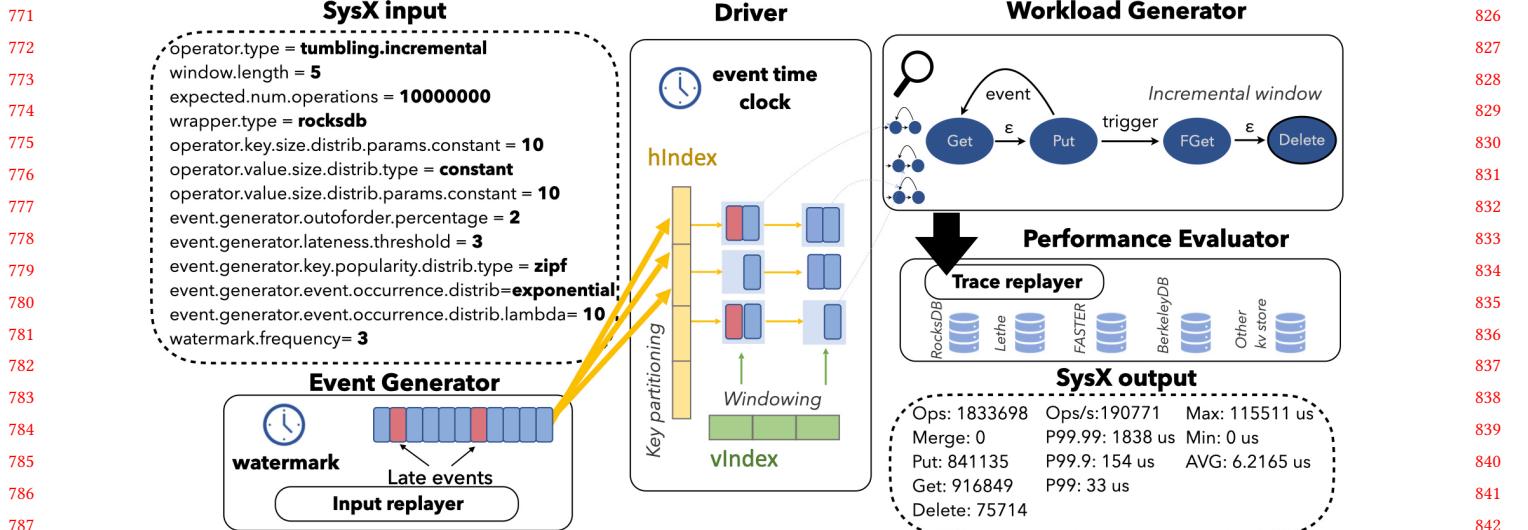


Figure 8. SysX architecture overview. The event generator generates event streams according to a configuration file. SysX’s driver partitions the input stream horizontally and/or vertically, depending on the logic of the specified operator. Each partition is assigned to a state machine. SysX’s workload 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.

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 (5s 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 SysX benchmark harness

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

We implemented SysX in C++ in 18K 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 SysX 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).

SysX 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, SysX 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, SysX maintains an index of late events that is updated at every time step.

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

Algorithm 1: SysX driver logic

```

881 1 Function driver():
882   // Pull and process the next batch of events
883   2 batch = getNext();
884   3 for event e in batch do
885     4 stateMachines = assignStateMachines(e);
886     5 for m in stateMachines do
887       6   | m.run();
888       7 end
889     8 end
890   9 Function onWatermark():
891   10 stateMachines = collectExpiredStateMachines();
892   11 for m in stateMachines do
893     12   | m.terminate();
894   13 end
895
896
897
898

```

64-bit timestamps to events, which can then be replayed using different time units. This allows *SysX* 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. *SysX*’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 *SysX* 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

```

936 switch(state) {
937   case PutState:
938     generateOp(PUT);
939     state = GetState;
940     done = TRUE;
941     break;
942   case GetState:
943     generateOp(GET);
944     trigger ? state = DeleteState : state = PutState;
945     break;
946   case DeleteState:
947     generateOp(DELETE);
948     done = TRUE;
949     break;
950 }

```

Figure 9. State machine for incremental tumbling window.

information is sufficient to generate accurate state access streams while keeping *SysX*’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 incurs a pair of get-put requests to retrieve and update the aggregated value associated with the event key.

SysX 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 *SysX* driver.

5.4 Extending *SysX* with new streaming operators

Algorithm 1 shows the driver logic in *SysX*. 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, *SysX* 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.

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 *SysX*-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 *SysX* include four types of operations $p = \{\text{get}, \text{put}, \text{merge}, \text{delete}\}$, 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. Adding a new kv store to *SysX* requires implementing a C++ wrapper that performs this translation.

6 Evaluation

Our evaluation is structured into three parts. In § 6.1, we empirically show that *SysX* 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 *SysX* workloads for evaluation, the throughput and latency results are close to those obtained with real traces. Finally, in § 6.3 we use *SysX* to evaluate the performance of four kv stores for eleven streaming workloads.

Experimental setup. We run all experiments on a dual-socket machine equipped with 12-core Intel Xeon 4116 CPU running at 2.1 GHz, 32GB of RAM, and a 512GB 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 128MB write buffers (memtables) and a 64MB cache. We further set the Lethe delete threshold to 10s. We use the B⁺Tree version of BerkeleyDB with a 256MB cache. The FASTER log and hash index use 256MB and 64MB respectively. We use the default values for all other configuration options. We repeat all experiments at least three times and report mean values.

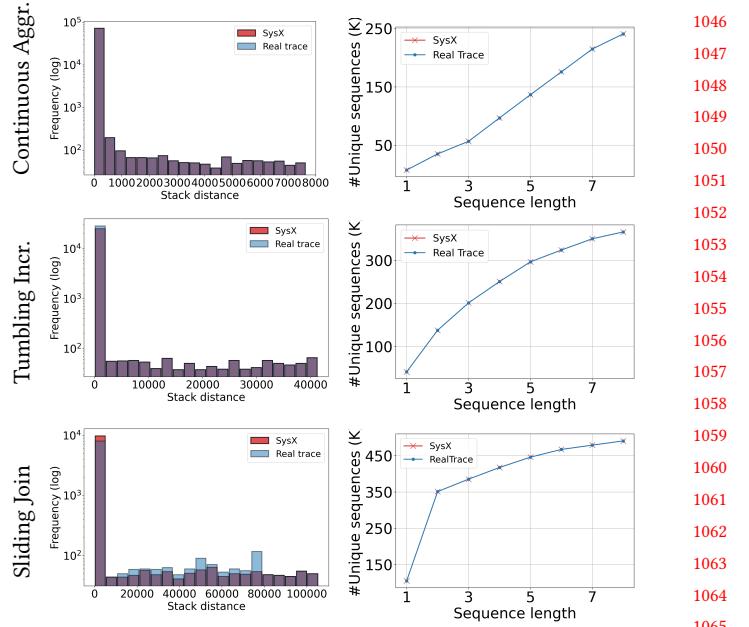


Figure 10. Stack distances and unique sequences in *SysX* and real workloads. *SysX*'s traces exhibit very similar spatial and temporal locality with the real traces.

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

In this section, we show that *SysX* faithfully simulates streaming state accesses and can produce workloads that exhibit the characteristics of real traces. We configure *SysX* 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 *SysX* 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, *SysX* produces a trace that consists of an almost identical number of unique sequences as the real trace. The distribution of stack distances in *SysX* 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, *SysX* 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 *SysX* can generate workloads that exhibit the same degree of temporal and spatial locality as real traces.

1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100

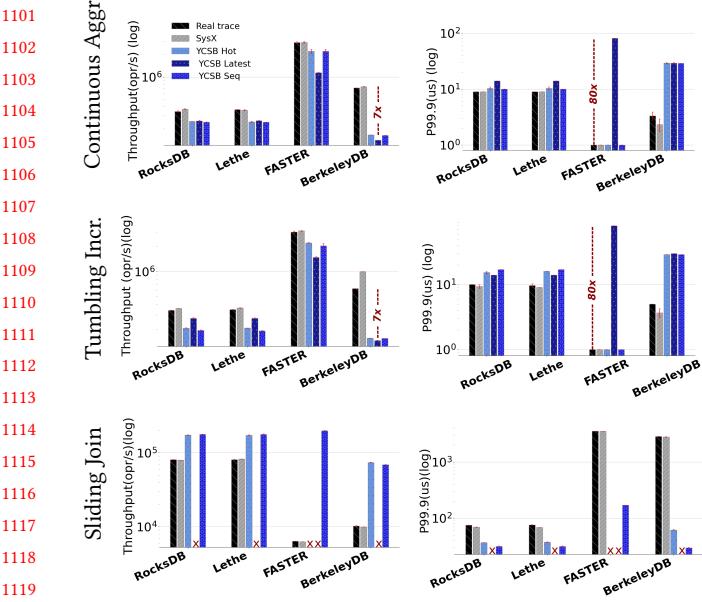


Figure 11. Throughput and latency measured with *SysX*, YCSB, and real traces. *SysX* results are close to those obtained with real traces. On the contrary, when using manually tuned YCSB workloads, the reported performance differs by up to an order of magnitude.

6.2 Are *SysX* workloads valuable in practice?

While *SysX* 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, 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 *SysX* traces to drive experiments with all four kv stores. We replay the workloads with the *SysX* 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 *SysX* 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

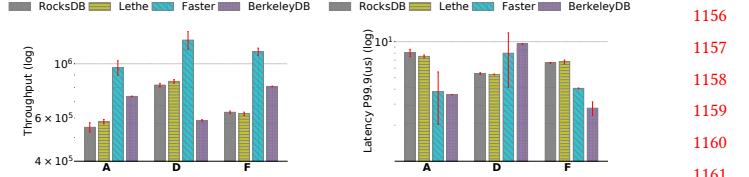


Figure 12. Performance evaluation results for all kv stores in *SysX* using three core YCSB workloads.

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 *SysX* traces.

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

6.3 *SysX* in action: Evaluating streaming state stores

Finally, we demonstrate *SysX* in action to evaluate the suitability of four kv stores for streaming state management.

Without access to *SysX*, 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).

Next, we repeat the experiment using all *SysX* 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.

1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210

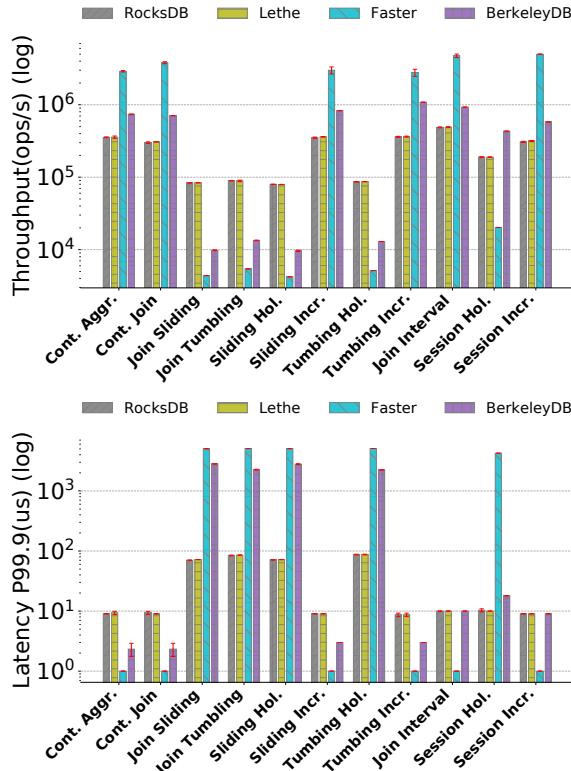


Figure 13. Performance evaluation of streaming state stores using SysX. RocksDB is outperformed in six out of eleven workloads but offers robust performance for all operators.

Overall, RocksDB and Lethe provide *robust* results: their tail latency does not exceed 100 μ s for any workload and remains below 10 μ s in many cases. If we can only select a single store for streaming state management, RocksDB is indeed the most sensible choice available today. However, our results suggest that 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 properties of the input stream and operator, 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 SysX. Other benchmarks, such as LinkBench [18] and BigDataBench [51], do not consider temporal patterns and, thus, cannot generate realistic streaming 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 SysX reveal many interesting opportunities for future work. SysX 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 Mill-Wheel [13] and Pravega [10], rely on distributed kv stores. We believe that SysX 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.

1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320

1321 References

- 1322 [1] Alibaba Realtime Compute. <https://www.alibabacloud.com/product/realtimecompute>. Last access: October 2021. 1376
- 1323 [2] Amazon Kinesis. <https://aws.amazon.com/kinesis/>. Last access: October 2021. 1377
- 1324 [3] Apache Flink. <https://flink.apache.org/>. Last access: October 2021. 1378
- 1325 [4] Azure Stream Analytics. <https://azure.microsoft.com/en-us/services/stream-analytics/>. Last access: October 2021. 1379
- 1326 [5] Berkeley DB. <https://www.oracle.com/database/technologies/related/berkeleydb.html>. Last access: October 2021. 1380
- 1327 [6] Google Cloud Dataflow. <https://cloud.google.com/dataflow>. Last access: October 2021. 1381
- 1328 [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. 1382
- 1329 [8] Kafka Streams Internal Data Management. <https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management>. Last access: October 2021. 1383
- 1330 [9] NYC TLC Trip Record Data. <https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page>. Last access: October 2021. 1384
- 1331 [10] Pravega. <https://pravega.io>. Last access: October 2021. 1385
- 1332 [11] RocksDB. <https://rocksdb.org/>. Last access: October 2021. 1386
- 1333 [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. 1387
- 1334 [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 VLDB Endowment*, 6(11):1033–1044, Aug. 2013. 1388
- 1335 [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. 1389
- 1336 [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, 1390
- 1337 [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. 1391
- 1338 [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. 1392
- 1339 [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. 1393
- 1340 [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. 1394
- 1341 [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. 1395
- 1342 [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. 1396
- 1343 [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. 1397
- 1344 [23] L. Bol'Shev and N. Smirnov. Tables in mathematical statistics. *Tables in Mathematical Statistics [in Russian]*, 1965. 1398
- 1345 [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. 1399
- 1346 [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. 1400
- 1347 [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. 1401
- 1348 [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. 1402
- 1349 [28] B. Chandramouli, G. Prasad, 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. 1403
- 1350 [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. 1404
- 1351 [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. 1405
- 1352 [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. 1406
- 1353 [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. 1407
- 1354 [33] M. Dayaratna and S. Perera. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, 51(2):1–36, 2018. 1408
- 1355 [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. 1409
- 1356 [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. 1410
- 1357 [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. 1411

- 1431 [37] V. Kalavri and J. Liagouris. In support of workload-aware streaming
1432 state management. In *12th USENIX Workshop on Hot Topics in Storage
1433 and File Systems (HotStorage 20)*. USENIX Association, July 2020.
- 1434 [38] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and
1435 V. Markl. Benchmarking distributed stream data processing systems.
1436 In *34th IEEE International Conference on Data Engineering, ICDE 2018,
1437 Paris, France, April 16-19, 2018*, pages 1507–1518. IEEE Computer Society,
1438 2018.
- 1439 [39] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M.
1440 Patel, K. Ramasamy, and S. Tanuja. Twitter heron: Stream processing at
1441 scale. In *Proceedings of the 2015 ACM SIGMOD International Conference
1442 on Management of Data*, pages 239–250, 2015.
- 1443 [40] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics
1444 and evaluation techniques for window aggregates in data streams.
1445 In *Proceedings of the 2005 ACM SIGMOD international conference on
1446 Management of data*, pages 311–322, 2005.
- 1447 [41] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation
1448 techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117,
1449 1970.
- 1450 [42] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow.
1451 In *CIDR*, 2013.
- 1452 [43] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha,
1453 A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, et al. Turbine: Face-
1454 book's service management platform for stream processing. In *2020
1455 IEEE 36th International Conference on Data Engineering (ICDE)*, pages
1456 1591–1602. IEEE, 2020.
- 1457 [44] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst,
1458 I. Gupta, and R. H. Campbell. Samza: stateful scalable stream process-
1459 ing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645,
1460 2017.
- 1461 [45] R. Pitchumani, S. Frank, and E. L. Miller. Realistic request arrival
1462 generation in storage benchmarks. In *2015 31st Symposium on Mass
1463 Storage Systems and Technologies (MSST)*, pages 1–10, 2015.
- 1464 [46] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces:
1465 format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.
- 1466 [47] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A
1467 Tunable Delete-Aware LSM Engine. In *Proceedings of the 2020 ACM
1468 SIGMOD International Conference on Management of Data, SIGMOD '20,*
1469 page 893–908, New York, NY, USA, 2020. Association for Computing
1470 Machinery.
- 1471 [48] U. Srivastava and J. Widom. Flexible time management in data stream
1472 systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-
1473 SIGART symposium on Principles of database systems*, pages 263–274,
1474 2004.
- 1475 [49] Q.-C. To, J. Soto, and V. Markl. A survey of state management in big
1476 data processing systems. *The VLDB Journal*, 27(6):847–872, Dec. 2018.
- 1477 [50] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark—A Bench-
1478 mark for Queries over Data Streams. Technical report, OGI School of
1479 Science & Engineering at OHSU, 2002.
- 1480 [51] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi,
1481 S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench:
1482 A big data benchmark suite from internet services. In *2014 IEEE 20th
1483 International Symposium on High Performance Computer Architecture
1484 (HPCA)*, pages 488–499, 2014.
- 1485 [52] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield. Charac-
1486 terizing storage workloads with counter stacks. In *11th {USENIX} Sympo-
1487 sium on Operating Systems Design and Implementation ({OSDI} 14)*, pages
1488 335–349, 2014.
- 1489 [53] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Char-
1490 acterizing storage workloads with counter stacks. In *11th USENIX Sympo-
1491 sium on Operating Systems Design and Implementation ({OSDI} 14)*, pages
1492 335–349, Broomfield, CO, Oct. 2014. USENIX Association.
- 1493 [54] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang.
1494 Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th
1495 ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '16*, New York,
1496 NY, USA, 2016. Association for Computing Machinery.
- 1497 [55] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds
1498 of in-memory cache clusters at twitter. In *14th USENIX Symposium
1499 on Operating Systems Design and Implementation (OSDI 20)*, pages
1500 191–208. USENIX Association, Nov. 2020.
- 1501 [56] S. Zheng, M. Hoseinzadeh, and S. Swanson. Ziggurat: A tiered file
1502 system for non-volatile main memories and disks. In *17th USENIX
1503 Conference on File and Storage Technologies (FAST 19)*, pages 207–219,
1504 Boston, MA, Feb. 2019. USENIX Association.
- 1505 [57] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash
1506 by using less cache. In *Proceedings of the 4th USENIX Conference on Hot
1507 Topics in Cloud Computing, HotCloud'12*, page 3, USA, 2012. USENIX
1508 Association.
- 1509 [58] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash
1510 by using less cache. In *Proceedings of the 4th USENIX Conference on Hot
1511 Topics in Cloud Computing, HotCloud'12*, page 3, USA, 2012. USENIX
1512 Association.
- 1513 [59] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces:
1514 format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.
- 1515 [60] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A
1516 Tunable Delete-Aware LSM Engine. In *Proceedings of the 2020 ACM
1517 SIGMOD International Conference on Management of Data, SIGMOD '20,*
1518 page 893–908, New York, NY, USA, 2020. Association for Computing
1519 Machinery.
- 1520 [61] U. Srivastava and J. Widom. Flexible time management in data stream
1521 systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-
1522 SIGART symposium on Principles of database systems*, pages 263–274,
1523 2004.
- 1524 [62] Q.-C. To, J. Soto, and V. Markl. A survey of state management in big
1525 data processing systems. *The VLDB Journal*, 27(6):847–872, Dec. 2018.
- 1526 [63] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark—A Bench-
1527 mark for Queries over Data Streams. Technical report, OGI School of
1528 Science & Engineering at OHSU, 2002.
- 1529 [64] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi,
1530 S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench:
1531 A big data benchmark suite from internet services. In *2014 IEEE 20th
1532 International Symposium on High Performance Computer Architecture
1533 (HPCA)*, pages 488–499, 2014.
- 1534 [65] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Charac-
1535 terizing storage workloads with counter stacks. In *11th {USENIX} Sympo-
1536 sium on Operating Systems Design and Implementation ({OSDI} 14)*, pages
1537 335–349, Broomfield, CO, Oct. 2014. USENIX Association.
- 1538 [66] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang.
1539 Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th
1540 ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '16*, New York,
1541 NY, USA, 2016. Association for Computing Machinery.