In support of *workload-aware* streaming state management

Vasiliki Kalavri* vkalavri@bu.edu
John Liagouris*† liagos@bu.edu
*Boston University, †Hariri Institute for Computing

Abstract

Modern distributed stream processors predominantly rely on LSM-based key-value stores to manage the state of long-running computations. We question the suitability of such general-purpose stores for streaming workloads and argue that they incur unnecessary overheads in exchange for state management capabilities. Since streaming operators are instantiated once and are long-running, state types, sizes, and access patterns, can either be inferred at compile time or learned during execution. This paper surfaces the limitations of established practices for streaming state management and advocates for configurable streaming backends, tailored to the state requirements of each operator. Using *workload-aware* state management, we achieve an order of magnitude improvement in p99 latency and 2x higher throughput.

1 Introduction

Any non-trivial streaming computation maintains and continuously updates state: rolling aggregations, synopses, window contents, triggers and timers. To support larger-than-memory state, streaming dataflow systems rely on data partitioning and (embedded) persistent key-value stores. The most prominent store is RocksDB [4], used by open-source systems such as Apache Spark Structured Streaming [9], Apache Flink [13], Apache Kafka [2], and Apache Samza [27], as well as Facebook's Stylus [17]. RocksDB has been widely adopted by many stream processors due to its solid performance on SSDs, incremental checkpointing capabilities, robustness, and active community. However, it was not designed with streaming workloads in mind.

We argue that the state management capabilities general-purpose stores provide to streaming applications come at considerable cost. The established *monolithic* approach to streaming state management is problematic: one type of state store (either RocksDB or in-memory) manages the state of all dataflow operators. Some operators, such as joins, accumulate large state and benefit form efficient range scans, while others, such as rolling counters, store small state and need efficient in-place updates. Yet, state requests are served by stores which are configured in a manner oblivious to each operator's state types, sizes, and access patterns.

With this paper, we advocate for *workload-aware* state management. The central idea is to exploit the fact that streaming dataflow operators are instantiated once and are long-running. Thus, their access patterns and state size bounds are largely known in advance. We believe that streaming systems would greatly benefit from novel configurable state stores, supporting different physical layouts and data types, and capable of leveraging knowledge about operator state characteristics.

As a proof of concept in support of workload-aware state management, we build a testbed that allows defining custom state stores per operator and we use it to compare state management strategies. As an alternative to RocksDB, we experiment with FASTER [15], a recent key-value store backed by a cache-optimized hash index with efficient in-place updates and support for custom data types. Our testbed is implemented on top of Timely dataflow [19], a high-performance streaming dataflow engine written in Rust.

To quantify the benefits of workload-aware state management, we compare RocksDB and FASTER on window operators and stateful queries from a streaming benchmark suite. Our evaluation shows that *not one store fits all* when it comes to streaming workloads. Although FASTER's in-place updates favor some queries, RocksDB's lazy evaluation is superior for others. Using FASTER instances with explicit type information and tailored to the needs of individual operators, we achieve an order of magnitude improvement in p99 latency and 2x higher throughput.

2 State management in streaming dataflows

We first revisit the fundamentals of dataflow stream processing and then describe established practices for streaming state management. The dataflow model described below applies with minor variations to the majority of distributed stream processors supporting data-parallelism and local state, such

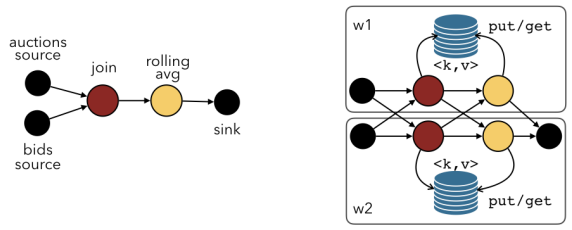


Figure 1: Logical and physical dataflow plans. Each parallel worker executes one or more stateful tasks and manages their state via an embedded k-v store instance.

as Apache Flink, Kafka, Samza, and Timely dataflow. Stream processors with external state management are briefly discussed in Section 2.2.

2.1 The dataflow programming model

A dataflow stream processor executes data-parallel computations on shared-nothing architectures. A program is represented as a *logical* directed graph $G = (V, E)$, where vertices in V denote operators and edges in E are data dependencies. Upon deployment, the logical graph is translated to a *physical* execution plan, $G' = (V', E')$, which maps dataflow operators to provisioned workers. We call vertices in V' *task*s of an operator in V and edges in E' physical data *channels*. One or more tasks of the same or different operators can be assigned to the same worker, and the assignment strategy is system-specific. The assignment is computed at deployment time and remains static throughout job execution, unless a reconfiguration occurs. Tasks are scheduled once and are long-running.

Figure 1 shows the logical graph and corresponding physical plan for Nexmark Q4 [3, 31]. The query joins a stream of auctions and a stream of bids and outputs a rolling average of winning bid prices for each auction category. The provisioned workers, $w1, w2$, execute parallel tasks as indicated by the enclosing rectangles, and each owns an embedded store to manage state. The join and average operators are stateful and *data-parallel*, so that each worker operates on *disjoint* partitions of the input streams.

2.2 State management

A streaming application can define and maintain different types of state throughout its lifetime, including partial results, sketches, buffered tuples in window buckets, triggers, timers, and notifications. A *state backend* defines the physical location where streaming state resides. The backend type determines how state is organized into data structures, its maximum supported size, and whether it is guarded against failures. For instance, an in-memory backend keeps working state in memory and, thus, the maximum supported state size is limited. A database backend keeps working state in an embedded key-value store and leverages the disk to support larger-than-memory state.

Users interact with state through APIs provided by the stream processor. Those allow defining *state primitives* of different types, such as lists, maps, and arbitrary values. Most dataflow programming models assume a key-value schema for input records [9, 13, 14, 20, 25] and always associate state with a *key*. The API also enforces *state scopes*, which determine the accessibility of state through the application.

The assignment of tasks to worker threads shown in Figure 1 has significant implications to state scoping. The runtime guarantees that input records with the same key will be processed by the same task. As a result, any state associated with a particular key is read and modified by a single worker at any point in time. By employing data parallelism, the dataflow model provides *single-writer isolation to state*. Thus, it is crucial that any key-value store used as a streaming backend provides excellent single-thread performance.

External state management. An alternative to managing state inside the stream processor is essentially making all operators stateless and externalizing all intermediate data. While this design simplifies fault-tolerance and re-configuration, it induces higher latency and presumes access to a highly available and scalable external state store. Some systems that follow this approach include Storm [30], which can spill state to HDFS or Cassandra [6], and MillWheel [8] that uses BigTable [16]. For further details, we refer interested readers to a recent survey on streaming state management [29].

3 Flaws of monolithic state management

Stream processors use embedded stores as out-of-the-box solutions for state management. This straw man approach restricts applications to using one type of backend, which is configured globally for all dataflow operators, regardless of their state requirements[1]. In this section, we summarize the drawbacks of this monolithic design.

3.1 Unnecessary data marshalling

An evident issue with storing multiple operator states in the same store is the requirement to manage keys and values of different types. To support arbitrary types, RocksDB maintains all in-memory and disk-resident data as byte arrays. This eager de/serialization approach means that *every* state access includes one or more data marshalling operations, all of which are on the critical path of the request. A recent study [23] finds that Flink spends 20% of execution time de/serializing RocksDB data of non-primitive types (e.g., `Vec<String>`), while similar results are shown in other works

[1] Flink's recent release (1.10) provides *some* flexibility, as it allows users to partially configure RocksDB column families (one per state primitive).

as well [7]. Samza tries to mitigate this overhead by maintaining a cache [28].

While a well-designed cache policy can avoid some data marshalling, we believe this task must be the responsibility of the state store. State types in streaming dataflow applications are known at compile time and do not change during execution. If each primitive is backed by a dedicated store, in-memory data can be kept in their native format, incurring de/serialization only when fetching or flushing pages from/to disk, asynchronously.

3.2 Oblivious store configuration

The second major issue with monolithic state management is that all state stores in a streaming job use the same configuration, regardless of the access patterns and size requirements of the operators they serve. This is problematic, as operators in a dataflow can have vastly different behavior in terms of state. For example, in the query of Fig. 1, the join is *write-heavy* and can potentially accumulate large state, while the aggregation performs two *read-modify-write* operations per input event and its state can fit in memory. A write-optimized store capable of spilling state to disk is suitable for managing the join state, while a hash-based in-memory store with in-place updates would fit the rolling aggregation better.

3.3 Unnecessary store features

Streaming backends do not need all sophisticated features of general-purpose key-value stores. In case we use one backend per operator task or state primitive, there is no need to support concurrent external requests within the backend, since dataflow systems already guarantee single-thread access to state. Although concurrency control does not affect single-thread performance in practice, it introduces unnecessary complexity and maintenance overhead. Similar simplifications are also possible for other features of existing key-value stores. For instance, the performance of deletes can be significantly improved by leveraging the knowledge of streaming operators. In the case of windows, the backend can simply purge the entire window state when it expires.

The following operations are handled by dataflow systems already and need not be implemented in state backends:

State partitioning is performed according to the input data partitioning functions used by shuffling operators.

State scoping is enforced by the dataflow state APIs, which abstract store operations and provide access to per-key local state only.

Per-key read/write isolation is guaranteed by the dataflow model. Worker threads process disjoint data partitions, thus, each key will always be accessed by a single thread.

Checkpointing is coordinated by the streaming runtime which uses locks and synchronization mechanisms to copy the state. Stream processors implement global snapshot algorithms [13] and do not rely on key-value stores for fault-tolerance.

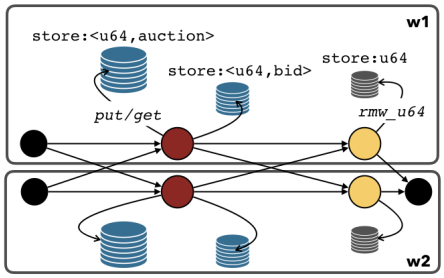


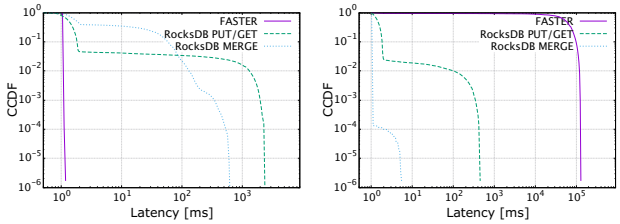
Figure 2: With *flexible state management*, each task can define multiple state stores of different types and configurations. Each instance of the join operator has two backends, one for each of its input streams.

4 A flexible testbed for state management

To explore the potentials of a more flexible state management approach, we have implemented a testbed on top of the Timely Dataflow stream processor [19]. Timely is a Rust implementation of Naiad [26] and allows for a fair performance evaluation, free from runtime and JNI overheads or other incompatibility issues present in JVM-based systems. For example, Flink's JNI bridge to RocksDB imposes a maximum key-value pair size of 2.2GB [2]. This limit may be too restrictive in some cases. A window operator can represent its state as a key-value pair, where the key is the start or end timestamp and the value contains the window contents, whose size may exceed 2.2GB (cf. Section 5.1). Further, custom merge comparators are not supported in Java, as RocksDB expects C++ code. Custom merge comparators are used by RocksDB for lazy evaluation in the `merge` operator. Merging is semantically equivalent to a Read-Modify-Write operation: each time a key-value pair is added to the database using `merge`, the pair is appended to the mutable mem-table and the actual value update is performed "lazily" during compaction (if any) or upon a `get` for the respective key, by calling the comparator function. Our testbed does not have any of the above restrictions, it minimizes the FFI overhead when interacting with the key-value store, and allows defining custom RocksDB comparators directly in Rust.

Currently, the testbed supports three types of backends: in-memory, RocksDB, and FASTER. We use a wrapper around FASTER's C++ library [1] to expose its interface for accessing the key-value store and performing internal operations. For the integration with RocksDB, we use the Rust wrapper [5]. We also provide a versatile API so that users can

[2] https://ci.apache.org/projects/flink/flink-docs-release-1.10/ops/state/state_backends.html#the-rocksdbstatebackend



(a) 30s-1s sliding window COUNT (b) 30s tumbling window RANK

Figure 3: RocksDB vs FASTER latency CCDFs for (a) a sliding window of 30s length and 1s slide with incremental aggregation and (b) a tumbling window of 30s length with holistic aggregation.

define basic state primitives in their streaming dataflows, including `ManagedCount` (for counters), `ManagedValue` (for arbitrary values), and `ManagedMap` (for maps) [11].

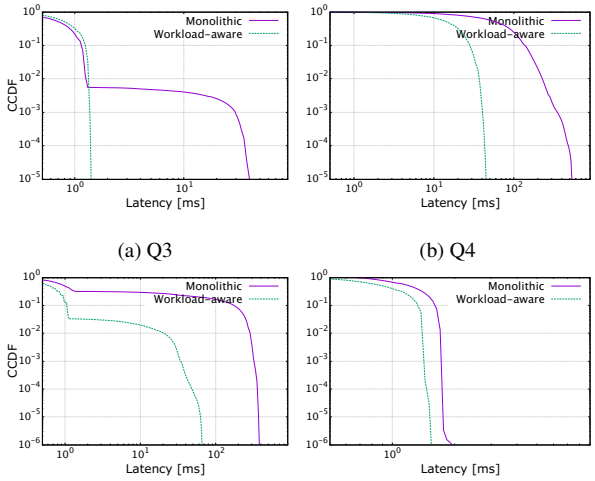
Figure 2 illustrates how users can take advantage of flexible state management with our API and testbed. Each operator can define one or more individually configured state stores, instantiated with specific types. Store configuration is currently manual, but we are working on automating this process by leveraging knowledge about the computation.

5 Experimental evaluation

We present a set of evaluation results with RocksDB and FASTER using our state management testbed[3]. The evaluation does not aim to thoroughly compare RocksDB with FASTER but to showcase that *not one store fits all* streaming workloads. We demonstrate the effect of a backend's data layout when evaluating window operators in § 5.1 and the effect of leveraging knowledge about types, state sizes, and state accesses in § 5.2. The evaluation is focused on tail latency, as the major performance metric for real-time streaming applications, and on single-thread performance, as the dataflow model guarantees per-key read/write isolation (§ 3.3). To provide a clear picture of tail latency, we use *complementary* CDFs (CCDFs): a (x, y) point indicates that y% of records have at least x *ms* end-to-end latency. Thus, the lines at $y = 10^{-1}$ and $y = 10^{-2}$ correspond to p90 and p99 latency, respectively. We briefly discuss throughput and multi-worker dataflows in § 5.2.

Benchmarks. We use the Nexmark streaming benchmark [31, 32]. Queries include an incremental join (Q3), window joins (Q4, Q6, Q8), and custom window aggregations (Q5, Q7).

Experimental setup. We use Timely `0.9.0`, compiled with Rust `1.37.0`, the latest FASTER C++ version from [1], and the RocksDB Rust wrapper (`0.12.4`) [5]. We configure FASTER with a 128MB hash index and 8GB in-memory log, where 10% (resp. 90%) is given to the immutable (resp. mutable) region, as in the FASTER paper [15]. RocksDB is configured with 128KB block size, 2 mem-tables of 4GB each, a 256MB LRU cache, and a 100MB hash block index. We use `c5d.2xlarge` and `r5d.12xlarge` instances on Amazon EC2, for single-thread and multi-thread experiments, respectively.



(a) Q3 (b) Q4

(c) Q5 (d) Q7

Figure 4: Latency CCDFs for monolithic vs. workload-aware implementation of Nexmark queries using FASTER.

5.1 The effect of data layout on windows

Any streaming state management backend needs to efficiently support windows, as they are perhaps the most common streaming operators. Windows enable evaluation of blocking computations, such as aggregations and joins on streams, and provide continuous fresh results to applications. For the purpose of this paper, we consider fixed windows which can be identified by their start and end timestamps. Windowing splits unbounded inputs into a series of bounded sets of records and we say that a window *triggers* when the system's notion of time arrives at its end timestamp. At this point, the window produces results and deletes its state. Window evaluation functions can be applied eagerly, upon receiving a new record that belongs to the window, or lazily, on trigger.

We evaluated various strategies and window functions and concluded that there is no clear winner between LSM-based and hash-based approaches. Nevertheless, we identified the parameters which affect performance and can guide the design of a configurable, workload-aware store. Our results indicate that FASTER performs better than RocksDB across configurations for eager aggregations, especially for large sliding windows. As for holistic aggregations, RocksDB's `merge` performs best across configurations and, it is the only approach that can keep up with high input rates. In the interest of space, we discuss only one representative experiment.

We evaluate `RANK`, a holistic aggregation that fetches the entire window contents on trigger, and `COUNT`, which can be

[3] Available at https://github.com/jliagouris/wassm

Query	RocksDB	FASTER	Workload-aware	Speedup
Q3	1.36	1.32	1.27	1.04
Q4	357.62	230.18	36.90	6.24
Q5	490.65	286.12	20.52	13.94
Q6	307.69	209.78	37.10	5.65
Q7	1.85	1.7	1.39	1.22
Q8	1.49	1.32	1.31	1

Table 1: p99 latencies (*ms*) for RocksDB, FASTER, and workload-aware implementations of Nexmark queries. The rightmost column shows the speedup of workload-aware over FASTER.

eagerly computed to maintain a single value per window. We run each experiment for 10 minutes in an open loop and measure the end-to-end latency per record. The input rate was set to 1K rec/s for `RANK` and 10K rec/s for `COUNT`. In both operators, the state is organized as follows: the key is the window start timestamp and the value is the window content (a vector of integers in `RANK` and a single integer in `COUNT`).

Figure 3 plots the latency CCDF for a sliding and a tumbling window. For every input record, the operator performs one read-modify-write operation for each active window state[4]. While FASTER supports in-place updates, RocksDB issues either a pair of `get` + `put` operations or a `merge`, depending on the implementation. On trigger, the operator retrieves the window contents with one `get` operation and then purges state. FASTER performs best for `COUNT` (100× lower p99 latency than RocksDB MERGE) as it can leverage fast lookups with in-place updates. For `RANK`, FASTER and RocksDB with PUT/GET continuously remove and reinsert growing vectors of integers, while lazy evaluation with RocksDB MERGE pays off (p99 latency is orders of magnitude lower than FASTER's).

5.2 State types, sizes, and access patterns

To showcase the additional benefits of workload-aware store configuration, we use carefully configured implementations of Nexmark queries that leverage knowledge about (i) k-v types (to reduce de/serialization), and (ii) state size and access patterns. FASTER was superior to RocksDB in all these queries, thus, our implementations use FASTER; in particular, one instance per primitive configured according to Table 2. The monolithic baseline uses one FASTER instance per worker configured as described in the experimental setup. We use 1M rec/s input rate for Q3, Q7 and Q8, and 10K rec/s for Q4-6.

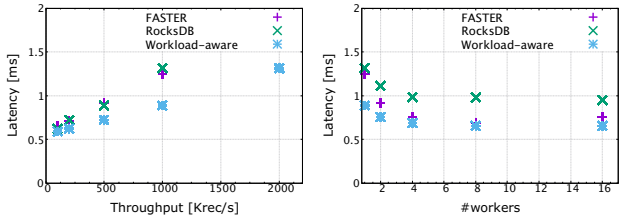
The first set of experiments evaluates single-thread latency for monolithic and workload-aware configurations. Then, we evaluate scaling with respect to the input rate (throughput) and the number of parallel workers.

Single-thread latency. Figure 4 shows the latency CCDFs for Q3, Q4, Q5, and Q7 run with a single worker thread (Q6 has almost identical behavior to that of Q4, and Q8 accumulates

[4]Note that several windows can be active (open) at the same time, as stream processors support asynchronous execution and out-of-order streams.

Query	Configuration
Q3	1GB (people), 7GB (auctions)
Q4	6GB (bids), 1.5GB (auctions), 512MB (average)
Q5	6GB (additions), 1GB (deletions) 512MB (accumulations), 512MB (hot items)
Q6	6GB (bids), 1.5GB (auctions), 512MB (average)
Q7	6GB (pre-reduce), 2GB (all-reduce)
Q8	4GB (people), 4GB (auctions)

Table 2: Memory given to FASTER instances in workload-aware implementations of the Nexmark queries from [32].



(a) Median latency vs throughput (b) Median latency over #workers

Figure 5: Q7 latency with monolithic and workload-aware state management over (a) throughput and (b) varying number of workers.

negligible state). Table 1 summarizes the p99 latency and speedup for all queries and state management approaches. The workload-aware implementation achieves significant p99 latency speedups for many queries: 14× for Q5 and 6× for Q4, Q6. Q3 has a 10× speedup as well, for higher percentiles.

To understand the source of performance improvements, we describe how our workload-aware implementations compare to the monolithic ones. Primitives that can grow arbitrarily and accumulate larger-than-memory state are backed by a dedicated FASTER store that is instantiated with the respective data types. For example, each of the two join inputs in Q3 (people and auction streams) has its own FASTER instance. The instances are typed after the stream they manage and their configuration is shown in Table 2. In Q3, most of the available memory is allocated to the auctions state, as auctions accumulate faster than people. As shown in Fig. 4a, this optimization pays off for the tail latency, where we see more than 10× speedup. The benefit is more evident for Q4 (Fig. 4b) and Q7 (Fig. 4d). In Q5, we split state further and use two more primitives with dedicated FASTER backends for auction counts and hot items (cf. Table 2). Small state with bounded size, such as the computation progress in Q8, is kept in memory.

Throughput vs latency. Fig. 5a plots the median per-record latency for Q7 for increasing input rates. We vary the input rate from 100K rec/s up to 2M rec/s, on a single thread. The latency of the workload-aware implementation scales better with increasing throughput and remains below 1ms for up to 1M rec/s. Further, with the workload-aware implementation, a single thread can comfortably sustain a 2M rec/s input rate with a median latency of 1.3ms. That is 2× higher throughput

than the monolithic approaches, which cannot keep up with input rates over 1M rec/s.

Multiple worker threads. To confirm that the benefits of workload-aware implementation persist in multi-worker dataflows, we run Q7 with an increasing number of worker threads (1-16), in an open-loop setting, with input rate fixed at $1M$ rec/s. Note that the dataflow model still ensures single-thread isolation to state, however, the number of key-value store instances sharing the resources of the same machine increases significantly. The monolithic approach uses one FASTER instance per worker, whereas the workload-aware uses two (cf. Table 2). Fig. 5b plots the median per-record latency. The workload-aware implementation scales similarly to the monolithic and remains consistently better as we increase the number of workers.

6 Conclusion

Workload-aware streaming state management can boost performance, beyond avoiding data marshalling costs. A careful assignment of state backends to primitives not only reduces latency but also improves throughput and scalability. Our results motivate the need for configurable streaming backends, in the spirit of the recent developments on designing data structures tailored around a particular workload [21, 22].

7 Discussion

One store that fits all or many? An open question towards implementing workload-aware state management for a production-ready stream processor is whether it is worth designing a new stream-optimized key-value store from scratch. One could argue that a selected set of existing key-value stores can be plugged-in via a flexible API like the one we discuss in Section 4. It certainly appears that a store like RocksDB could serve range queries and lazy evaluation whereas a store like FASTER could be used for operators with frequent point lookups and in-place updates. However, this approach introduces undesirable project dependencies for the streaming system and does not account for changes in state characteristics (e.g. due to increased input rates) that might make pre-configured backends unsuitable over time.

Streaming state benchmarks. Despite the fact that key-value stores are an integral component of streaming systems and crucial to their performance, their suitability for streaming state management has not been studied prior to this paper. In fact, no benchmark exists that captures the workload characteristics and temporal locality of streaming applications. Existing key-value store evaluations [10, 15, 24] focus on multi-threaded performance and use request-driven benchmarks, such as YCSB [18], which is oblivious to key-space locality [12]. Such benchmarks and metrics cannot provide reliable results for data-parallel stream processors, which instead execute continuous queries, perform single-thread state access, and issue frequent deletions (e.g. for windows).

Fault-tolerance and re-configuration. Even though stream processors that implement their own checkpointing mechanisms do not rely on key-value stores for providing exactly-once guarantees upon failures, certain store features are still desirable. Besides reads and writes in regular processing, a streaming engine must also support efficient state operations for fault-tolerance and re-configuration. Those include quick copies, incremental checkpointing, state migration upon re-partitioning, and bulk loads for quick recovery. For example, although RocksDB supports incremental checkpoints (at the SST level) by leveraging key order, hash-based stores, such as FASTER, do not. How to support all these operations efficiently within the same key-value store is an open question.

Extracting and/or learning state characteristics. The queries of Section 5 use built-in operators, however, most real streaming computations include operators with UDFs. One option to infer state access patterns in UDFs is to use static code analysis. This approach can be further combined with online learning techniques to infer data-dependent access patterns and changes in state characteristics. Learning access patterns with temporal dependencies in a streaming setting is an interesting direction for future research.

References

[1] FASTER. https://github.com/microsoft/FASTER. Last access: March 2020.

[2] Kafka Streams Internal Data Management. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management. Last Access: March 2020.

[3] Nexmark Benchmark Suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/. Last access: March 2020.

[4] Rocksdb. https://rocksdb.org/. Last access: March 2020.

[5] RocksDB Rust Wrapper. https://github.com/rust-rocksdb/rust-rocksdb. Last access: March 2020.

[6] Apache Cassandra. http://cassandra.apache.org, 2020. Last access: March 2020.

[7] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 113–119, New York, NY, USA, 2019. ACM.

[8] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proceedings of the the VLDB Endowment*, 6(11):1033–1044, August 2013.

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei 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.

[10] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, Renton, WA, July 2019. USENIX Association.

[11] Matthew Brookes, Vasiliki Kalavri, and John Liagouris. Faster state management for timely dataflow. In *Proceedings of Real-Time Business Intelligence and Analytics*, pages 1–3. 2019.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. 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, February 2020. USENIX Association.

[13] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, August 2017.

[14] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter 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.

[15] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike 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.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceeding of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[17] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell 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.

[19] Frank McSherry. Timely Dataflow. https://github.com/TimelyDataflow/timely-dataflow. Last Access: March 2020.

[20] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment*, 12(9), 2019.

[21] Stratos Idreos and Tim Kraska. From auto-tuning one size fits all to self-designed and learned data-intensive systems. In *ACM SIGMOD*, 2019.

[22] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*, pages 535–550. ACM, 2018.

[23] Gyewon Lee, Jeongyoon Eo, Jangho Seo, Taegeon Um, and Byung-Gon Chun. High-performance stateful stream processing on solid-state drives. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, pages 9:1–9:7, New York, NY, USA, 2018. ACM.

[24] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[25] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[26] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *SOSP '13: Proceedings*

of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013.

[27] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[28] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017.

[29] Quoc-Cuong To, Juan Soto, and Volker Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27(6):847–872, December 2018.

[30] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm @Twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD '14)*, 2014.

[31] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark—A Benchmark for Queries over Data Streams. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[32] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark Benchmark Suite, 2002.