FlowKV: A Semantic-Aware Store for Large-Scale State Management of Stream Processing Engines

Gyewon Lee
FriendliAI
gyewonlee@friendli.ai

Jaewoo Maeng
Jinsol Park
Seoul National University
{jwmaeng,jinsolpark}@snu.ac.kr

Jangho Seo
NAVER Corp.
jangho.seo@navercorp.com

Haeyoon Cho
Seoul National University
hy.cho@snu.ac.kr

Youngseok Yang
Mirny Inc.
ys@mirny.io

Taegeon Um
Samsung Research
taegeon.um@samsung.com

Jongsung Lee
Samsung Electronics
Seoul National University
leitia@snu.ac.kr

Jae W. Lee
Seoul National University
jaewlee@snu.ac.kr

Byung-Gon Chun*
FriendliAI
Seoul National University
bgchun@snu.ac.kr

Abstract

We propose FlowKV, a persistent store tailored for large-scale state management of streaming applications. Unlike existing KV stores, FlowKV leverages information from stream processing engines by taking a principled approach toward exploiting information about *how* and *when* the applications access data. FlowKV categorizes data access patterns of window operations according to how window boundaries are set and how tuples inside a window are aggregated, and deploys customized in-memory and on-disk data structures optimized for each pattern. In addition, FlowKV takes window metadata as explicit arguments of read and write methods to predict the moment when a window is read, and then loads the tuples of windows in batches from storage ahead of time. Using the NEXMark benchmark as workload, our experiments show that Apache Flink on FlowKV outperforms Flink on RocksDB or Faster with up to 4.12× throughput gain.

CCS Concepts: • **Information systems** → **Stream management**.

Keywords: stream processing, KV store, state management

ACM Reference Format:
Gyewon Lee, Jaewoo Maeng, Jinsol Park, Jangho Seo, Haeyoon Cho, Youngseok Yang, Taegeon Um, Jongsung Lee, Jae W. Lee, and Byung-Gon Chun. 2023. FlowKV: A Semantic-Aware Store for Large-Scale State Management of Stream Processing Engines. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567493

1 Introduction

Stream processing engines (SPEs) such as Apache Beam [17], Apache Flink [12], and Apache Samza [64] process infinite streams of timestamped key-value (KV) tuples. At the heart of SPEs lie stateful window operations that group incoming data streams into finite windows [2]. States of window operations were traditionally stored in memory to ensure high throughput and low latency. However, the growing amount of data has prompted recent studies [4–6, 10, 29, 36, 44, 82] and industrial products [11, 35, 42, 75, 85] to handle states that exceed the capacity of the memory. To this end, modern stream processing applications manage their states on fast storage (e.g., NVMe-enabled solid state drives) using persistent KV stores [22, 23, 25, 70].

For example, the widely used SPEs, Flink and Samza, use and support RocksDB [32] for such use, and some studies [9, 10, 44] experimentally use Faster [13] as their KV store backend. However, existing persistent KV stores are not built with stream processing in mind. Hence, using them to handle window operations leads to the following issues.

First, existing SPE KV store backends deal with state stores in a monolithic manner, regardless of the multiple data access patterns of window operations [44]. Different window operations require different data access patterns, such as reading the value of a key (`Get()`), adding or updating the value of a key (`Put()`), appending new values to the values of an existing key (`Append()`), and reading the values of a range of keys (`Scan()`). Unfortunately, no one-size-fits-all persistent KV store can efficiently support all window operations, as demonstrated in a recent work [4].

Second, using SPEs with existing KV stores suffers from high CPU and I/O overhead even for competitive options (i.e., Faster for operations with `Get()` and `Put()`, RocksDB for operations with `Append()`). This is because existing KV stores do not leverage information that stream processing operators provide, such as when a certain data will be read, or what kind of operations are actually needed within the stream processing environment. Unaware of when the data

*Corresponding author

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567493

will be read, existing KV stores waste CPU cycles or I/O bandwidth for future read requests that are not likely to happen. Furthermore, they often adopt synchronization mechanisms that are not necessary for stream processing, where the keys are partitioned among single-threaded workers.

We propose FlowKV, a novel composite store for SPEs that leverages the semantics of streaming applications for efficient state accesses on persistent storage. We classify the window operations according to their data access patterns, and design three customized stores that are optimized to fully exploit window semantics of different data access patterns (Append and Aligned Read (AAR), Append and Unaligned Read (AUR), and Read-Modify-Write (RMW)). Specifically, each store is designed to exploit information about *how* and *when* window operations access data.

Leveraging *how* for customized data layout. FlowKV equips each store with customized data layouts that make use of window semantics. Window operations are categorized by the way they aggregate data and read the aggregates, which FlowKV derives from the functions of window operations. Based on this information, FlowKV chooses to deploy a certain store equipped with in-memory and on-disk data layouts that are customized to efficiently handle the data access pattern of the window operation. Specifically, unlike existing KV stores that organize their values based merely on the keys, FlowKV uses an in-memory write buffer that hashes data based on window boundaries or on a tuple of key and window boundaries. This unique hashing method directly leads to customized on-disk log file management, where log files are created per-window or as a global log file with an index file. Such designs help FlowKV efficiently handle a window operation's unique data access pattern.

Leveraging *when* for predictive batch read. FlowKV also leverages *when* data is read to prefetch data that are likely to be read in near future. Prefetching is a widely used technique in many database systems by using information such as spatial locality or index data to decide on which data to prefetch. However, prefetching in FlowKV is different from previous prefetching methods, in that it exploits window semantics to choose which data to prefetch. Specifically, FlowKV uses an algorithm that exploits information on window operations to *decide* or *predict* the read time. The latter termed *predictive batch read*, predicts read times by combining runtime data (i.e, timestamps of incoming data tuples and the windows that the tuples belong to), and statically defined semantics (i.e, window size, session gap) of window operations. This design greatly contributes to increasing throughput and overcoming excessive CPU overhead of existing persistent KV stores without increasing latency.

We evaluate the end-to-end query performance of Flink deployed on FlowKV upon AWS EC2 instances. Running the NEXMark queries [18] with diverse state access patterns, our evaluation shows that FlowKV achieves higher throughput compared to RocksDB and Faster by up to 4.12× and 3.45×, respectively. This validates that FlowKV is an adequate KV store for multiple data access patterns. Furthermore, the decrease of store CPU and I/O overhead is the main contributing factor to the overall execution time reduction, proving how FlowKV efficiently exploits window semantics that is not considered by previous KV stores.

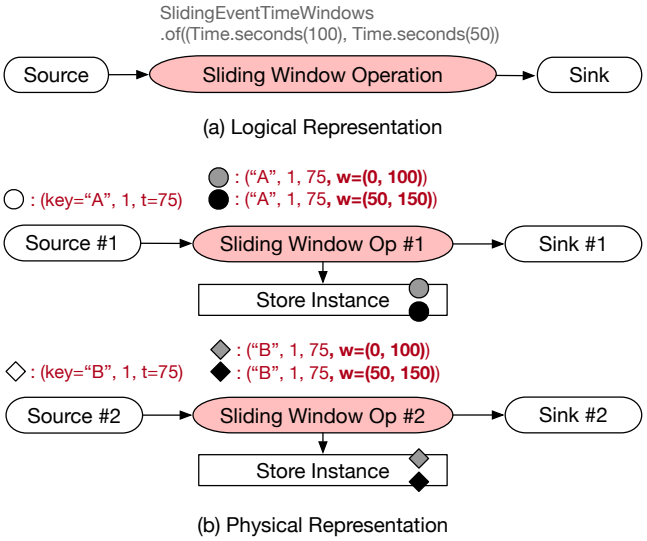

Figure 1. The logical and the key-partitioned physical representations ($parallelism = 2$) of an example streaming application with a sliding window operation whose window size is 100 seconds and sliding interval is 50 seconds. `t` and `w` indicate timestamp and window, respectively.

We summarize our contributions as follows.

- We empirically analyze the CPU and I/O bottlenecks introduced by persistent KV stores when the KV stores are used to handle the states of streaming applications.
- We design and implement FlowKV to solve this problem. It utilizes information from SPEs to exploit how and when window operations access data by (1) employing customized data layouts that use window boundary information, and (2) predicting estimated read times for efficient prefetching using window information.
- Using the NEXMark benchmark, we evaluate the end-to-end query performance of Flink on FlowKV. Compared to Flink on other widely used persistent KV stores, Flink on FlowKV achieves up to 4.12× throughput gain while maintaining comparable tail (95th-percentile) latency to using existing persistent KV stores as backends.

2 Background and Motivation

We first describe the semantics and data access patterns of window operations in stream processing. We then explain in detail the limitations of existing KV stores integrated with SPEs to maintain large internal states.

2.1 Stream Processing and Windows

Streaming applications running on SPEs continuously process infinite streams of data that consist of timestamped key-value tuples $e = (k, v, t)$, where $k \in K$ denotes a key, K denotes the key space, v denotes a value, and t denotes a timestamp. These applications are typically written in a high-level dataflow programming model such as the Flink programming model [12] and Beam [17], and are transformed to logical plans that are represented as directed acyclic graphs (DAGs). A logical plan (Figure 1(a)) consists of vertices that represent stream processing operations that transform timestamped key-value tuples, and edges that indicate data dependency. Before being deployed, a logical plan is compiled to the corresponding physical plan (Figure 1(b)) to execute the logical operations in parallel. Inside a physical plan, each logical operation is partitioned into multiple physical operations $p_0, ..., p_{n-1}$ where n denotes the degree of parallelism. p_i $(0 \leq i < n)$ processes tuples belonging to a non-overlapping key partition $K_i \subset K$ and $\bigcup_{i=0}^{n-1} K_i = K$.

Stream processing operations are largely divided into stateless and stateful operations. Stateless operations process input tuples one by one and send the result to downstream operations. Stateful operations, on the other hand, maintain internal states to perform more complicated operations such as windowing. In order to reduce communication overhead, many modern SPEs maintain their states locally and create a separate store instance for each physical operator p_i [12, 64], as illustrated in Figure 1(b). Thus, states are not shared among different streaming applications and physical operators and are accessed by a single-threaded worker.

Stateful operations are mostly window operations, which collect input tuples into finite data groups called *windows* [2] and aggregate them. A window can generally be defined with the start time and the end time $(start_W, end_W)$, each of which denotes the foremost and backmost time boundary, respectively. SPEs assign each tuple to eligible windows by adding a $(start_W, end_W)$ tag to the tuple [2, 21]. As described in Figure 1(b), if a tuple is assigned to two or more windows SPEs replicate the tuple and store each of the replicated tuples separately [2].

The window operation is defined by its aggregate function and its window function, both of which are given via high-level programming models of SPEs. The aggregate function defines how the collected tuples inside each window are aggregated. When the internal timer of the window operation reaches the end time for a window, the window operation aggregates tuples in the window using the aggregation function and sends the results to downstream operations. We refer to this as the window being *triggered*. The window function defines how to split unbounded streams into bounded windows by assigning a set of corresponding windows for a given tuple.

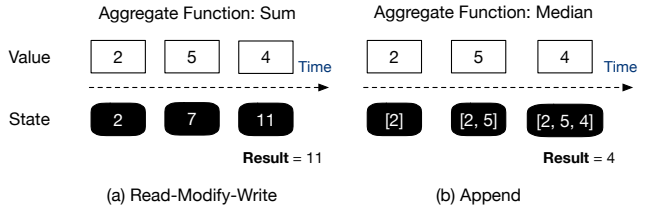

Figure 2. Aggregate functions determine whether to perform (a) a read-modify-write, or (b) an append operation.

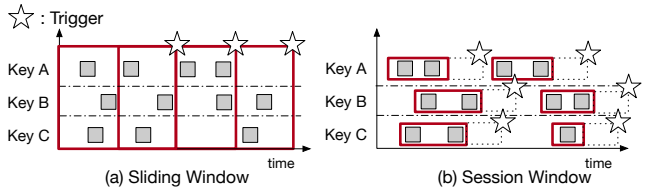

Figure 3. Window functions determine whether to read (a) all keys, or (b) each key when a window is triggered.

The way a window operation reads and writes its internal states is determined by its aggregate and window functions. We explain how the data access patterns of window operators can be categorized according to the different types of aggregate and window functions.

Aggregate Functions Aggregate functions determine how to update the internal states of window operations when new tuples arrive. We categorize aggregate functions depending on how to update internal states on tuple arrival.

- **Read-Modify-Write** If the aggregate function is associative and commutative, the operation applies incremental aggregation [77] as an optimization. With incremental aggregation, the operation may only maintain its intermediate aggregates instead of the entire list of windowed tuples. For example, Figure 2(a) shows that an aggregate function that computes sums is applied incrementally for the values.
- **Append** If the aggregate function is not associative or not commutative, the operation should maintain a list of collected tuples $[e_0, ..., e_{n-1}]$ for each window as its internal state. When a new tuple arrives, the operation appends the new tuple e_n to the end of the list of tuples in the window. For example, Figure 2(b) shows that an aggregate function that computes medians operates on the entire list of values that were previously appended. Another example is the windowed join, whose operation is blocked until all the tuples within a window are gathered.

Window Functions Window functions determine how to read the stored data. We categorize window functions according to whether the windows are aligned in time for all the keys in the key space K or not.

- **Aligned Read** Window functions such as fixed and sliding window functions create windows with a fixed length for every sliding interval. This allows the operation

to simply read data associated with all the keys in the key space K once the end of a window has reached. For example, Figure 3(a) shows that windows of all the keys are triggered and read at the same time.

- **Unaligned Read** For session windows, the window boundaries are determined separately for each individual key, after a period of inactivity (called *session gap*) for this key. The session window functions group a period of continuous activity over a key-partitioned data stream, delimited by a session gap that exceeds a predetermined time limit. In such cases, the read is unaligned in terms of how the data may be read at different times according to their keys. For example, Figure 3(b) shows that the boundaries of session windows are determined dynamically, and triggered on a per-key basis. Specifically, for a key-partitioned sub-stream of key k, any tuples whose keys are k and occurrences are within a time interval less than the session gap g are captured inside a single window. If there is no tuple arrived for g, the collected tuples are aggregated according to the aggregate function.

FlowKV classifies stream operations as a combination of aggregate functions and window functions, and uses three stores (Append and Aligned Read (AAR), Append and Unaligned Read (AUR), and Read-Modify-Write (RMW)). We take this approach because streaming applications are mostly static: their semantics are pre-defined and do not change for a long time. The read patterns (aligned or unaligned) are not differentiated for the RMW aggregate function because operations read existing data upon every arrival of a KV tuple, which makes the read patterns irrelevant.

2.2 Limitation of persistent KV Stores

SPEs such as Flink [12] and Samza [64] by default use the in-memory store to maintain the window state. The in-memory store provides fast data access, but is limited by memory capacity. In the case of handling large window states, SPEs delegate state management to external persistent KV stores. In such cases, the assigned window and the key of the tuple are used as the key for the KV stores.

Broadly speaking, persistent stores can be categorized by whether they maintain sorted data structures (e.g., LSM-tree and B-tree) or not. RocksDB, which is the most widely used KV store for handling large states by SPEs [12, 64], uses LSM-tree to maintain data in sorted order. Other KV stores such as Faster [13] maintain non-sorted and partially-sorted data structures, to reduce CPU overhead. The sorted/non-sorted nature of these KV stores affects how they perform with different data access patterns of stream applications.

To understand the performance of KV stores with SPEs, we run Flink 1.9 [12], a popular open-source SPE on RocksDB and Faster, which are two commonly selected KV backends for modern SPEs [5, 6, 9–11, 29, 35, 36, 42, 44, 57, 62, 68, 75, 79] from the sorted and non-sorted domains, respectively. We

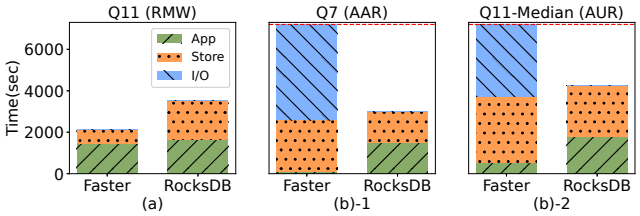

Figure 4. Execution-time breakdown of processing approximately 400GB of stream data with Flink on RocksDB and Faster. Red dotted lines indicate jobs that did not finish even after running for 7200s.

run three queries from the NEXMark Benchmark [18] (Q7, Q11-Median, Q11) that have three different combinations of data access patterns we described: Append and Aligned Read (AAR) pattern, Append and Unaligned Read (AUR) pattern, and Read-Modify-Write (RMW) pattern, respectively.

Figure 4 shows the execution time and its breakdown that are taken to process approximately 400GB stream data with Flink on different stores. We use perf [27] to gather CPU stack traces and generate a flamegraph [38], and leverage dstat [26] to extract I/O wait times. We then analyze the visualized stack traces of the flamegraph to obtain the breakdown as a result.

Figure 4 displays limitations of existing persistent KV stores when handling states of SPEs. First, existing KV stores are only efficient for certain types of window operations due to the diversity of their state access patterns. For example, between the two persistent KV stores, Faster outperforms RocksDB for RMW (Q11 in Figure 4 (a)), because Faster maintains an unsorted data structure that enables $O(1)$ access to a key-value pair, whereas RocksDB suffers from increased search overhead caused by maintaining key-sorted data structures. On the other hand, for window operations that invoke `Append()` (Q7, Q11-Median in Figure 4 (b)-1, and (b)-2), RocksDB shows better performance than Faster because RocksDB adopts lazy merging [33, 44], which first appends values to log files without reading existing values that then get merged later. Faster, which is optimized for in-place updates of key-value pairs, reads and writes all the previously appended values on every `Append()`, and thus incurs excessive I/O amplification. Therefore, for append workloads, Faster fails to process all the input data tuples even after running for 7200 seconds due to overwhelming I/O and system call overhead, at which point we terminate the job.

Second, even for their efficient operations (i.e., Faster for Q11, and RocksDB for Q7 and Q11-Median), the KV stores still suffer from significant CPU overhead that is close to the overhead of query processing. This is mainly because they do not leverage useful knowledge from SPEs (Figure 4). For example, when handling window operations that use `Get()` and `Put()`, Faster adopts synchronization mechanisms that are unnecessary for modern SPEs that maintain local states,

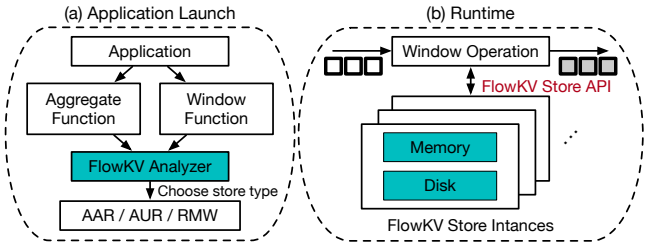

Figure 5. Overview of FlowKV. One of the three store patterns determined at application launch time decides the store API to be used through runtime, along with optimized data layouts on memory and disk.

each of which is accessed by a single thread [12, 64]. RocksDB periodically merges the appended values in order to handle random `Get()` requests that require low processing latency, which entails CPU overhead. However, it is possible to predict future window reads from window functions and real-time timestamps of tuples as we explain in § 4.2. Without leveraging such an opportunity, RocksDB cannot avoid high CPU overhead caused by frequent merging.

We summarize our key observations below.

- There is no one-size-fits-all persistent KV store that can cover all the data access patterns efficiently, since each pattern of the window operations requires different optimization strategies.
- Even the competitive KV store for each data access pattern still suffers from high CPU overhead, spending CPU time similar to or more than query computation in SPEs. This problem is more prominent on append workload, which requires searching formerly appended items during lazy merging.

To overcome these limitations, we believe that a new composite KV store backend is needed for the following reasons. First, since streaming applications have diverse state access patterns, it is hard for a single KV store to efficiently handle all the access patterns. Second, existing KV stores do not leverage information from SPEs such as which window each tuple belongs to or the time boundary of each window, although such information provides valuable hints on future data accesses.

3 FlowKV Overview

FlowKV is a novel composite store for SPEs that exploits window semantics, especially the information about *how* and *when* window operations access data. Figure 5 shows the overview of FlowKV. At application launch, FlowKV determines the store pattern based on the aggregate function and window function. Additionally, FlowKV deploys separate FlowKV store instances according to the number of partitions. A physical window operator p_i is in charge of tuples whose keys belong to K_i, where K_i denotes a key space partition of K. FlowKV further partitions K_i into $K_{i,0}, ..., K_{i,m-1}$ and makes a FlowKV store instance $s_{i,j}$ process states whose keys belong to $K_{i,j}$. m is a configurable parameter of FlowKV and indicates the number of store instances deployed for each physical window operator. This design, transparent to users, reduces compaction overhead and latency spikes, as compaction is triggered for each store instance independently on its state partition.

At runtime, FlowKV uses specific APIs for the chosen store pattern to read and write data. Each FlowKV Store has customized data layouts optimized for its data access pattern. Also, it determines the window trigger time using algorithms that exploit window semantics. Next, we explain the details of how FlowKV determines the store patterns at application launch time (§ 3.1), and the APIs that FlowKV uses at runtime for each store pattern (§ 3.2).

3.1 Determining Store Patterns

Figure 5 (a) shows how FlowKV determines which store pattern to deploy at application launch using the aggregate and window function signatures. First, using the aggregate function signature, FlowKV determines whether the window operation has the RMW or the Append pattern by leveraging the fact that the aggregate function implements one of the pre-defined function interfaces provided by SPEs [12, 17, 64]. If the aggregate function implements the function interface that merges the new KV tuple into the intermediate aggregates (e.g., `AggregateFunction` in Flink), FlowKV determines the window operation has the RMW pattern. On the other hand, if the function implements the interface that requires all the KV tuples to be prepared before triggers (e.g., `ProcessWindowFunction` in Flink), FlowKV assumes that the window operation has the Append pattern.

Second, to determine whether the read pattern is Aligned or Unaligned, FlowKV maps window functions to their data access patterns. This is straightforward since each window function has its own predetermined read pattern. For example, fixed and sliding window functions create windows of fixed lengths for every sliding interval and therefore are mapped to the Aligned Read pattern. On the other hand, session and count window functions determine window boundaries dynamically and therefore are mapped to the Unaligned Read pattern. Custom window functions, for whose semantics are unknown, are assumed to have the Unaligned Read pattern that can cover both patterns.

3.2 FlowKV Store API

Listing 1 lists out the API functions that each type of FlowKV store (AAR, AUR, RMW) supports along with their description. FlowKV APIs are distinguishable from existing KV stores in that they receive runtime information specific to stream processing. Instead of taking mere K (key) and V (value) as input, all operations explicitly use W (window information) to exploit window semantics. Additionally, some

Listing 1. List of data access APIs of the three store types of FlowKV. K, V, W, T, and A stand for the types of key, value, window, timestamp, and aggregated result, respectively.

```
/* AAR Store */
Iterable<(K, List<V)> GetWindow(W) //
    Fetch & remove an iterable of key
    and list of values of the window
void Append(K, V, W) // Append the KV
    tuple with its window
/* AUR Store */
List<V> Get(K, W) // Fetch & remove key
    and list of values of the window
void Append(K, V, W, T) // Append the
    KV tuple with its window and
    timestamp
/* RMW Store */
A Get(K, W) // Fetch & remove an
    aggregated result of the key and
    window
void Put(K, W, A) // Put the updated
    aggregate with its window
```

operations also make use of other stream processing information such as T (timestamp), and A (aggregate result). Although FlowKV provides different APIs from traditional KV stores, it does not require modification of user-level streaming applications, because its API is only exposed to SPEs. We elaborate on how each store reads and writes data using stream processing-specific API.

Append and Aligned Read (AAR) For the AAR store, all the KV tuples belonging to the same window are read at the same time, regardless of their keys. Accordingly, to read, the AAR store uses the `GetWindow()` function, which takes W as its argument to find the target log file on the disk. All the keys belonging to the given window W and their corresponding values are returned as an iterable of tuples, each tuple holding a key and its corresponding values as a list. To write, the AAR store uses the `Append()` function. When the `Append()` function is called, the given W is used as a key for the in-memory hash table, and the given value V is appended to the key-value list in the correct hash bucket labeled by W. The in-memory buffer is flushed to the disk when it exceeds the user-given memory capacity limit. Apart from in-memory hashing, W is also used to track the trigger time of the given window, at which point all KV tuples within that window will be read.

Append and Unaligned Read (AUR) For the AUR store, the windows of each key are triggered at different points in time. Accordingly, when reading, the `Get()` function takes K and W as arguments, locates the data position in the global data file, and returns the corresponding values as a list. To write, the AUR store uses the `Append()` function like the AAR store. However, it additionally takes the timestamp as an argument, which is added to the log file along with the value to provide information when calculating the estimated read times for predictive batch read.

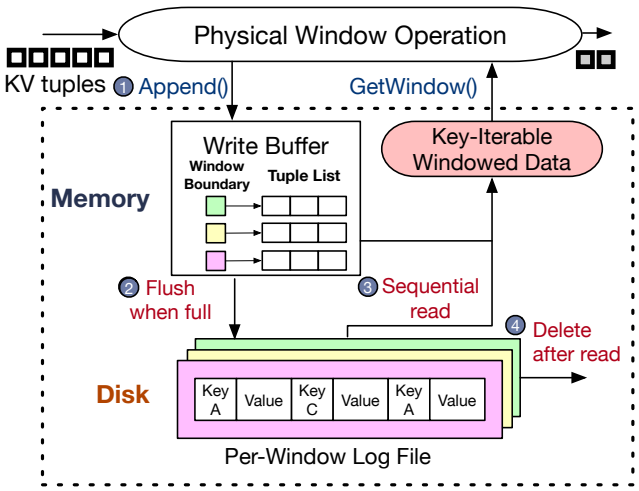

Figure 6. The append and aligned read store and its end-to-end state access procedure. The dotted line indicates boundary of FlowKV.

Read-Modify-Write (RMW) Unlike the previous stores that append new values, the KV tuples are aggregated to an existing aggregate value upon their arrival. Under such circumstances, `Get()` reads the data by hashing with (K, W), and returns a single aggregated value (A), instead of a list of values. After performing desired operations (e.g. addition), the updated value is written back using `Put()`. The `Put()` function takes K and W to hash the data based on the (K, W) tuple key, and updates the previous aggregated value with the new aggregated value, which is passed on as a function argument.

4 FlowKV Stores

This section dives into the details of each FlowKV store. For each store, we explain the state access procedures, and the details of leveraging *how* and *when* with customized data structures and predictive batch read.

4.1 Append and Aligned Read (AAR)

The AAR store exploits the fact that windows of different keys share identical trigger times and thus per-key access is not required.

State Access Procedures Figure 6 shows the overall structure and data access procedure of a single store instance for the append and aligned read. ① When a tuple (k, v, t) along with its window boundary $(start_W, end_W)$ arrives, `Append()` is called and a (k, v) pair is appended to the in-memory write buffer. The write buffer finds the corresponding key-value list by hashing the given window boundary $(start_W, end_W)$ and adds the (k, v) pair to the end of the list. ② If the size of a write buffer exceeds the user-given capacity limit, the buffer flushes its in-memory data to the end of the on-disk

log file, which is created per window. ③ When a window is triggered, `GetWindow()` is called multiple times to gradually read the corresponding log file. ④ Once data read is done, the per-window log files are cleaned from the disk.

Coarse-grained Data Organization FlowKV leverages *how* by using a coarse-grained form of data organization based on how the AAR store triggers multiple keys simultaneously. FlowKV organizes in-memory and on-disk data structures by window boundaries, not by their keys. This is different from naïve KV stores that manage data in a fine-grained manner (i.e., organize data by their keys).

In memory, a hash table within the write buffer organizes its data according to window boundaries $(start_W, end_W)$. For example, consider two tuples $e1 = (k1, v1, t1)$ and $e2 = (k2, v2, t2)$ that belong to the same window with a boundary $(start_W, end_W)$. Instead of making separate hash buckets for $k1$ and $k2$, both $(k1, v1)$ and $(k2, v2)$ are added to the list of the same bucket labeled $(start_W, end_W)$. If a tuple belongs to multiple window boundaries, SPEs replicate the tuple, and FlowKV stores the replicated tuples in each bucket.

On disk, each store instance maintains a separate data log file for each window boundary, namely for each hash bucket label of the write buffer. At data flush, data inside each hash bucket is appended to the end of the log files of the corresponding window boundaries, as shown in Figure 6. Using this design, FlowKV can assure that all the data inside the same log file will be read and removed at the same time. Hence, instead of searching through the logs looking for each key log file, FlowKV just looks for the file that keeps all the data of the given window boundary. Moreover, it can skip additional compaction processes and reduce the overhead latency, because each log file can simply be cleaned from the disk after being read.

Gradual State Loading For AAR, leveraging *when* is straightforward since window trigger times are determined in advance. However, loading all the data sharing the same window boundary at the same time may put heavy memory pressure. Therefore, in order to solve this problem and make the usage of coarse-grained data practical, FlowKV designs the `GetWindow()` method call to return only a partition of the window states loaded from a store instance. The SPE calls `GetWindow()` multiple times until it returns `null`, which indicates that there is no more partition to read. This design lets the SPE sequentially read and aggregate each partition so that only one non-aggregated partition exists in memory.

4.2 Append and Unaligned Read (AUR)

Dealing with window operations with the AUR pattern is more challenging since windows of different keys are read at different times. Hence, naïvely batching multiple window reads to reduce search overhead will result in latency increase, which is unacceptable in stream processing. Instead, the AUR store adopts *predictive batch read*, which predicts trigger times of windows based on the window function and timestamp of tuples, and reads windows in batches that are expected to be triggered in near future. By taking this approach, FlowKV can feed two birds with one scone, achieving high throughput via batching without latency increase.

State Access Procedures Figure 7 describes the structure and end-to-end state access procedures of the AUR store. ① When a tuple (k, v, t) arrives, `Append()` is called and the tuple is appended to the hash bucket of the corresponding window in the write buffer. Because each key of tuples maintains separate windows, windows inside the write buffer are identified by both tuple key *and* window boundary. Since window boundaries can dynamically change upon tuple arrival, FlowKV leverages the initial window boundary, which is defined and fixed when the window is first created. ② During `Append()`, FlowKV calculates the estimated trigger time (ETT) for the given window based on the timestamp t, and writes the updated ETT into the in-memory `Stat` table. ③ When the in-memory write buffer becomes full, the data inside the write buffer are flushed to the disk. During a flush, index entries (i.e., key, window metadata, offset, and length) are appended to the index log file and the tuple values are appended to the data log file.

④ When a `Get()` request for a key and a window comes in, FlowKV first searches through the in-memory prefetch buffer where the states read from predictive batch read are stored. If FlowKV cannot find the tuples for the given key and window, it reads the states of the target window from the disk, along with the states of other windows that are predicted to be triggered soon (predictive batch read). ⑤ To retrieve location information, FlowKV scans the index log file on disk, which contains the locations and lengths of each tuple, as well as the keys and the window metadata. FlowKV selects the indexes of N-smallest keys and windows ordered by time left until trigger to be loaded into memory. N is determined by multiplying the user-configurable prefetching ratio by the number of keys and windows. ⑥ Using the location information, the AUR store reads the disk blocks containing the eligible data and loads them into the prefetch buffer. The loaded states are organized according to their windows inside the prefetch buffer. If the ETT is correct for a window, the corresponding states are directly read from the prefetch buffer. If the estimation is wrong (e.g., a new tuple arrives before expiration in session window functions), the prefetched states are evicted from the buffer. ⑦ Compaction is integrated with predictive batch reads for I/O optimization.

On Disk Index Log File FlowKV leverages *how* by considering that for AUR, window boundaries are dynamically determined for each key. However, unlike the AAR store, maintaining a separate log file per window boundary is inadequate for the AUR store, because this approach will result in an excessive number of log files. Instead, the store puts the states of all the windows to the global append-only log. Hence, it is necessary to efficiently determine which parts of the global log should be read. This is challenging, however,

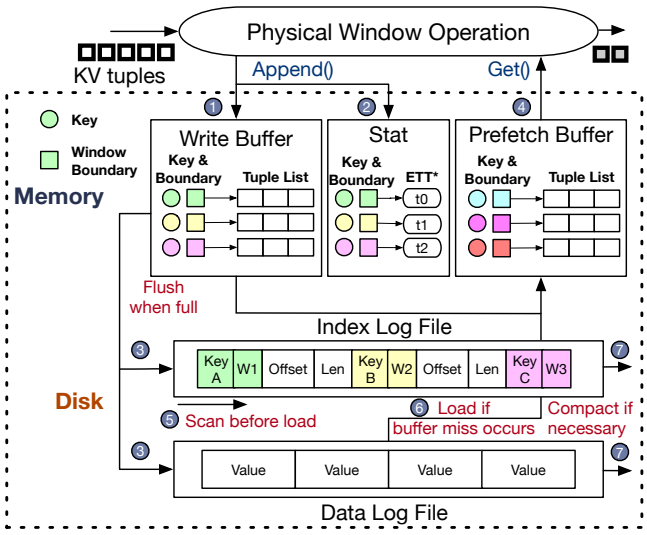

Figure 7. The append and unaligned read store and its end-to-end state access procedure. ETT stands for estimated trigger time. The dotted line indicates the boundary of FlowKV.

because indexes containing locations of states of a window can be too big to be stored in memory since the data of each window are scattered throughout the global log.

To this end, FlowKV creates on-disk indexes and updates the indexes in an append-only way. Upon flushing the write buffer, index entries (each of which consists of a key, window metadata, and location) are appended to the end of the on-disk index log file. Using an index log file minimizes memory overhead because on-disk indexes are loaded only when predictive batch read starts and only the locations of entries are loaded instead of the actual data. Even though the size of actual data is small, this approach is still effective because multiple KV tuples are merged into a single index entry inside the in-memory write buffer. Index search overhead can be minimized as well because locations of values belonging to multiple windows can be fetched with a single index scan.

Trigger Time Estimation FlowKV leverages *when* by predictive batch read for the AUR store. To estimate the future trigger time of each window with confidence, FlowKV leverages both the window function that is statically defined at application deployment and the timestamp of each tuple that is dynamically determined during runtime. Note that the statically determined information is often not sufficient, because some window functions (e.g., session window function) dynamically determine window boundaries with respect to incoming tuples. Based on the information, FlowKV calculates the ETT for each window. If the window function is one of the pre-defined functions whose semantics are already known, FlowKV automatically maps one of the pre-defined *predictors*. In the case of session window functions, for example, ETT can be calculated by adding the session gap to the maximum timestamp of the tuples within the window, assuming that no data would come during the session gap. If the window function is a custom function whose semantics are unknown, FlowKV may receive predictors from users (§ 8). ETTs are maintained as an in-memory hash table (Stat in Figure 7), and updated upon every tuple arrival.

Predictive batch read is effective when FlowKV can safely assume that each window will not be triggered before its ETT. In the case of session window functions whose session gap is g, for example, a window is guaranteed not to be triggered until $t_{max} + g$, where t_{max} denotes the maximum timestamp of the tuples inside the window. With this assumption, FlowKV can guarantee that buffer miss will not occur until all the prefetched window states are read or evicted. In some window functions, however, FlowKV cannot provide such a low bound for future triggers. This happens when the window functions are only based on the arrival of input data (e.g., the count window function). In such cases, it is difficult for our batched read to be effective because buffer misses may occur too frequently.

Selecting Windows To Be Read Based on ETTs, predictive batch read loads not only the requested state but also the states of windows that are expected to be triggered in near future into memory. FlowKV determines the windows to be loaded by selecting the N-smallest windows in the order of time left until ETT, assuming that such windows have a higher probability of being triggered in near future. Application developers can modulate N by setting the ratio of windows to be read in batches over all windows (*read batch ratio*) and thus adjust trade-off points between memory consumption and throughput. A detailed examination of trade-off points is described in § 6.4.

Predictive Batch Read Efficiency Predictive batch read aims to exploit an adequate trade-off point between CPU overhead and modern SSD I/O bandwidths. Modern SSDs offer significantly higher speed and lower latency compared to traditional storage devices, which provides us with extra I/O bandwidth resources [14, 28, 51, 65–67]. Predictive batch read uses this ample resource for less CPU computation. With KV stores that are competitive for `Append()` operations, such as RocksDB, the LSM tree structure is used to maintain key-sorted data structures. This structure enables individual lookups, used to only read the data wanted. However, although individual lookup allows fewer data read, it results in high search overhead with more CPU computation. With predictive batch read, reading in batches incurs linear scan across the log file, which can lead to more I/O, but significantly less CPU overhead. As shown in Figure 4 (b)-2, FlowKV handles an AUR requests using only 15% of the Store time spent for RocksDB. This is possible because FlowKV shows a high hit ratio thanks to its semantic-aware prefetching design with low read amplification.

When the ETT estimation is wrong, the cached tuples are evicted from the memory and need to be read again from the disk, which may cause read amplification. The number of

times a single tuple will be read can be calculated by adding all the possible cases of being read at once, twice, and so on. If a tuple is hit at an nth attempt, it would have been missed n-1 times before. Given a hit ratio r ($0 \leq r \leq 1$), this can be expressed with the equation below, resulting in the read amplification being inversely proportional to the hit ratio.

$$\sum_{n=1}^{\infty} nr(1-r)^{n-1} = \frac{1}{r} \quad (1)$$

We have found from the evaluation that using a read batch ratio of 0.02 results in a high hit ratio of around 0.93 (§ 6.4). Using Equation 1, it can be seen that each tuple is read only about 1.08 times. Considering the I/O bandwidth of modern SSDs, this amount of read amplification does not degrade the performance. Instead, FlowKV's CPU overhead reduction results in optimized performance.

Integrated Compaction The AUR store cannot simply delete the log file after the read because tuples of all the keys and windows are stored in global log files. For this reason, an individual compaction process, building new log files that contain only the valid data and removing the old log files, is necessary. The compaction process of FlowKV is different from that of existing KV stores in that FlowKV integrates the compaction process and predictive batch read. Instead of making separate index log scans for the predictive batch read and compaction process, we take advantage of the fact that the predictive batch read inevitably requires scanning the index log. Thus, when scanning the index log for predictive batch read, two things are checked for each item in the log. One is whether the current item is the index for what predictive batch read is aiming to fetch. The other is whether the tuple in the location indicated by the current item's index can be discarded. This eliminates multiple index log scans, which would have been necessary if predictive batch reads and the compaction process were not integrated.

FlowKV performs compaction when space amplification caused by the deleted data exceeds a user-given threshold called *maximum space amplification (MSA)*. This way, users can choose the trade-off point between disk space consumption and compaction overhead; a smaller MSA will help save disk space, whereas a larger MSA will reduce the overhead from frequent compactions. Note that not only the data log file but the index log file is also cleaned up during compaction. Since a FlowKV store instance is in charge of only the subsets of the states of the physical window operation, a single compaction process does not take much time and thus it does not severely harm the tail latency as shown in § 6.2.

4.3 RMW

The RMW store does not adopt read time prediction because `Get` should be always called upon every arrival of the tuple. Thus, the RMW store is similar to unsorted KV stores [13, 87], except that it does not support concurrent accesses. Each store instance maintains an in-memory hash write buffer and a hash index, as well as a log file. The hash indexes contain the on-disk locations of each tuple as well as its key and window, which can be used to quickly find the exact location of data. The compaction of the RMW store is done similarly to that of hash KV stores [13] when the space amplification exceeds the MSA.

5 Implementation

FlowKV is implemented with approximately 3K lines of Java 1.8 code. The current FlowKV implementation works with Flink 1.9.0.

Off-heap Data Management A FlowKV instance resides in a Java Virtual Machine (JVM) process that runs a Flink worker to eliminate inter-process communication overhead, such as memory copy between JVM on-heap memory and native memory. Therefore, the performance of FlowKV is affected by garbage collection (GC) overhead in JVM memory. To alleviate such GC pressure, each FlowKV store instance maintains its buffer inside off-heap regions that are not affected by the JVM GC process.

Zero-copy Byte Transfer Compaction in an AUR store is done by moving the valid part of a log file to a new one. To transfer bytes between the two log files, we leverage zero-copy file transfer to avoid unnecessary file-copy overhead between kernel memory and user memory. The AUR store can determine the exact location of bytes to be transferred in the data log files by scanning the index log files in advance. Thus, FlowKV store can invoke the zero-copy byte transfer system call with the file descriptors of the old and new data log files as well as the range of the data to be transferred.

6 Evaluation

We evaluate the performance of Flink, a popular open-source SPE, on FlowKV and other stores (the in-memory store, RocksDB, and Faster).

Evaluation Environment We run our evaluation on top of AWS EC2 instances. Three types of EC2 instances are used in our evaluation, for the following three roles. **Worker nodes** are the main instances for Flink workers that run the streaming applications. A worker node is deployed on an *i3.2xlarge* instance equipped with eight vCPUs, 61GB memory, a network up to 10Gbps, and a 1.9TB NVMe SSD. A **manager node** runs the Flink job manager for Flink worker management and Apache Kafka [47] message broker. The Kafka message broker is utilized for measuring the tail latency of each system in § 6.2, by constantly providing data streams with a fixed tuple rate. A **logger node** executes a source generator and a logger.

Workload For our evaluation, we use the NEXMark benchmark [18, 84], a popular workload that emulates online auctions and used to benchmark SPEs [17, 45, 52]. We do not use Gadget [4] for our evaluation, because we aim to measure the end-to-end streaming performance rather than the

sole performance of KV stores. In our evaluation, we evaluate queries that incur the access patterns of the three types of stores designed in FlowKV. We denote these queries as AAR, AUR, and RMW pattern queries throughout this section, meaning queries that incur append and aligned read, append and unaligned read, and read-modify-write patterns each. We use the following eight original and derived queries with various window semantics and state access patterns.

Q5 counts the number of bids per auction within sliding windows (RMW) and lists the auctions with the most bids using consecutive sliding windows (RMW). **Q5-Append** also counts the number of bids per auction within sliding windows (RMW) but finds the auction with the most bids without incremental aggregation (AAR). **Q7** counts the highest bid per bidder within fixed windows using side inputs, which enforces the append pattern by keeping KV tuples until they are triggered (AAR). **Q7-Session** is derived from Q7 by changing the fixed window function to the session window (AUR). **Q8** monitors new users who create a new auction within fixed windows, containing windowed join operations (AAR). **Q11** counts the number of bids each user makes within session windows (RMW). **Q11-Median** is derived from Q11 by replacing the count aggregate function with the non-associative median aggregate function (AUR). **Q12** counts the number of bids for each user within a global window, which indicates that all the tuples belong to a single window boundary (RMW). The other queries are excluded because they are not stateful (Q0, Q1, and Q2), do not contain window states (Q3), contain custom window functions whose access patterns FlowKV cannot identify (Q4, Q6, and Q9), or incur high overheads on window triggers (Q10). **Input dataset** We generate datasets for NEXMark queries using the data generator provided in Apache Beam [17]. Each KV tuple contains either person, auction, or bid information, and the average byte-serialized size of person, auction, and bid KV tuples is 16B, 16B, and 84B respectively. Data streams created by the generator consist of 2% person tuples, 6% auction tuples, and 92% bid tuples.

General Configuration We run a single Flink TaskManager on each worker node, having 16 Tasks (Working threads). The amount of memory assigned for TaskManager varies according to the storage systems. To evaluate the in-memory store, the majority of the memory (50GB) is allocated for TaskManager on-heap memory, where all the windows are stored. For Flink on FlowKV, 35GB of TaskManager memory is allocated as off-heap memory and 15GB of memory is allocated as on-heap memory, as off-heap memory is utilized as buffers. For queries that require more JVM heap memory due to a large number of keys or windowed joins (Q5, Q5-Append, and Q8), we allocate more JVM on-heap memory (40GB for Q5 and Q5-Append. 25GB for Q8). In cases of Flink on RocksDB and Faster, we allocate 16GB of memory to TaskManager, because RocksDB and Faster are C++ applications thus they do not share the memory with `TaskManager`. They utilize the remaining memory as their write buffers, in-memory indexes, and block caches.

FlowKV Configuration FlowKV has four configurable parameters: read batch ratio, write buffer size, maximum space amplification (MSA), and the number of partitions. We empirically set the read batch ratio to 0.02, the write buffer size to 2048MB, the MSA to 1.5, and the number of store instances (m) per physical window operation to 2. For queries containing sliding windows where each tuple is assigned to two windows (Q5, and Q5-Append), we double the write buffer size. We also analyze how such parameters impact the performance of FlowKV with microbenchmarks in § 6.4.

RocksDB and Faster Configuration We follow the official tuning guideline of RocksDB [34] and Faster [61] for configuration. For the evaluations of RocksDB, we allocate 2048MB of memory as write buffers, and the rest of the memory is allocated to in-memory block caches and the OS page cache. Faster configures the number of entries in the hash index and the total log size, which are set to 262144 and 1GB respectively for each Faster instance. Since Faster does not have Java APIs nor glue code for Flink, we implement such code on the C++ version of Faster.

6.1 Throughput on Changing State Sizes

We first analyze the performance of Flink on FlowKV as well as other stores by measuring their throughput on varying window sizes. To calculate throughput, we measure the time taken to process fixed-sized streaming datasets generated by the Beam NEXMark generator. We evaluate the eight queries with three different average window sizes and set the sliding interval to half of the window size for queries with sliding windows (Q5 and Q5-Append). If possible, state sizes are estimated using window size, access pattern knowledge, and the number of bytes processed provided by Flink. For example, if 1TB data is processed inside the JVM within 4000 seconds of event time, two fixed windows of size 2000 seconds would have been processed, making state size 500GB. Except for RMW queries, longer windows result in larger window states, since the number of tuples each window contains is proportional to the window length.

Figure 8 shows the measured throughput of each system. In summary, Flink on FlowKV performs better than Flink on RocksDB and Faster in most cases, demonstrating up to 4.12× throughput increase thanks to lower CPU time consumed during state access operations.

AAR Q7 and Q8 in Figure 8 show the throughputs of Flink on FlowKV and other stores when processing queries with the AAR pattern. In the case of Q7, Flink on FlowKV shows 1.87×–1.99× throughput increase compared to Flink on RocksDB, due to its ability to efficiently handle append operations without high search overhead. For the case of Q8, Flink on FlowKV shows 1.55×–1.65× throughput gain compared to Flink on RocksDB, and 3.45× throughput gain compared to Flink on Faster. The performance improvement

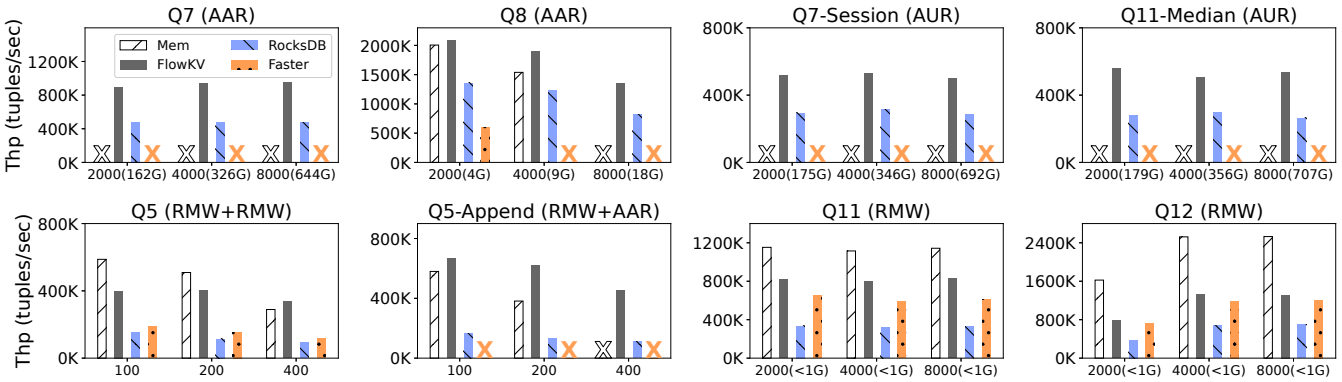

Figure 8. Throughput for the NEXMark queries with increasing window sizes with Flink on the in-memory store, FlowKV, RocksDB, and Faster. Crossed bars indicate failures due to out-of-memory errors or timeouts. The x-axis for each plot is window length in seconds (state size). It is difficult to estimate state sizes of workloads that use consecutive sliding windows, and therefore state sizes of Q5 and Q5-Append are omitted.

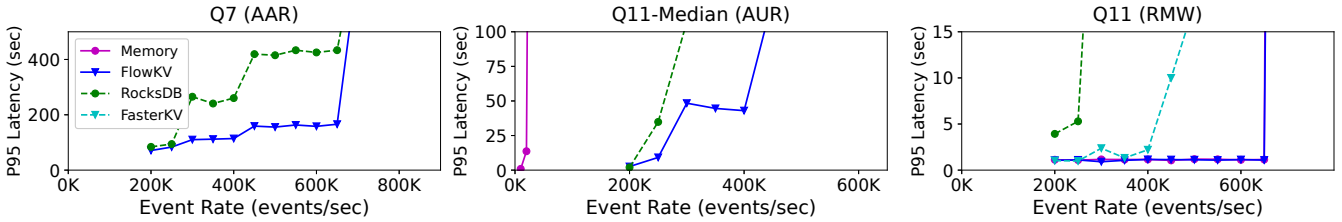

Figure 9. P95 latency graph of Q7, Q11-Median, and Q11 with 2000s window.

in Q8 is smaller compared to that of Q7 because Q8 maintains smaller states. We could not measure the throughput of Flink on the in-memory store for Q7, because the state sizes do not fit in memory. Additionally, Flink on Faster could not be measured, since Faster reads and writes all the appended values on every `Append()`, incurring excessive I/O.

Interestingly, in some points, Flink on FlowKV shows better performance compared to Flink on the in-memory store. This is because the in-memory store suffers from the JVM garbage collection (GC), which becomes severe as the state size increases. By efficiently leveraging persistent storage, FlowKV can reduce the JVM GC overhead.

AUR Q7-Session and Q11-Median in Figure 8 show the results of handling queries of the AUR pattern. Similar to the AAR queries, we could not measure the throughput of Flink on the in-memory store and Flink on Faster. Also similar to the case of queries with the AAR pattern, Flink on FlowKV and Flink on RocksDB show similar throughput due to the small state size on Q8-Session. For Q7-Session and Q11-Median, which have relatively larger state sizes, Flink on FlowKV shows 1.68×–2.02× performance compared to Flink on RocksDB.

RMW Q11 and Q12 in Figure 8 show the results of RMW queries. In Q11 and Q12, the throughput of Flink on FlowKV is 1.89×–2.52× higher compared to that of Flink on RocksDB because RocksDB experiences excessive search overhead due to its key-sorted data structures. In addition, Flink on FlowKV shows 1.27×–1.36× better performance compared to Flink on Faster. The benefit comes from consuming low CPU time for data access operations and avoiding unnecessary synchronization overhead.

Mixed Access Patterns Q5 and Q5-Append contain consecutive sliding window operations and thus have complex state access patterns. In these queries, Flink on FlowKV shows the highest performance gain, showing 2.64×–4.12× throughput compared to Flink on RocksDB and 2.13×-2.85× throughput compared to Flink on Faster. With consecutive window operations, performance benefit from FlowKV is accumulated over consecutive window operations. This result suggests that the effectiveness of FlowKV is maximized as the state access patterns become complicated.

6.2 Tail Latency

We evaluate the tail latencies of Flink on FlowKV and other stores on varying tuple rates. To measure latencies, we fix the incoming tuple rates for a certain period of time and measure the 95th-percentile (P95) latencies of each system. We ensure warm-up time by executing each system for the window size before measuring latencies. We show our evaluation results for the three queries (Q7, Q11-Median, Q11), each of which represents AAR, AUR, and RMW access patterns. For the other queries, we also observed the similar results.

Figure 9 shows the P95 latencies of each system with 2000s window. For Q11, Flink on RocksDB shows relatively higher latencies. Flink on Faster shows low latencies until the event rate reaches $400K/sec$, but fails to handle higher tuple rates.

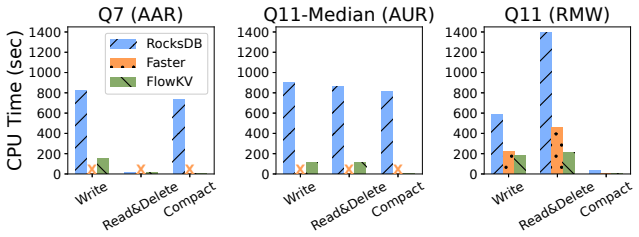

Figure 10. CPU times consumed by write, read & delete, and compaction operation in FlowKV, RocksDB, and Faster.

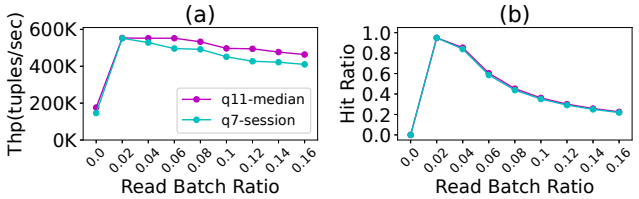

Figure 11. (a) Throughputs and (b) hit ratios of the prefetch buffer in Q11-Median and Q7-Session (window size=2000s).

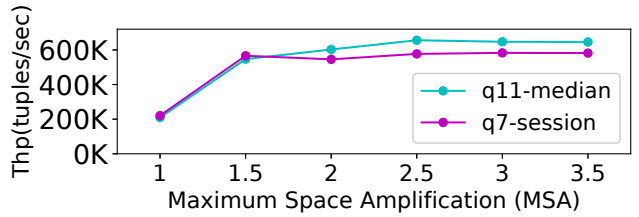

Figure 12. Throughput on different MSA configurations (window size=2000s).

Flink on FlowKV shows low latencies even for the higher tuple rates, comparable to Flink on the in-memory store. With Q11-Median, Flink on FlowKV shows tail latencies comparable to Flink on RocksDB at low tuple rates but can handle high tuple rates with lower latencies as well. Flink on the in-memory store fails early due to severe memory pressure. Flink on Faster also fails for all the tuple rates due to high CPU and I/O overhead in the append access pattern. In Q7, Flink on FlowKV shows similar or lower latencies to those of Flink on RocksDB at all tuple rates. Again, the in-memory store and Faster fail due to severe memory pressure and high CPU and I/O overhead. Results for Q7 show relatively high latencies for all working stores in general due to the bursty CPU overhead that comes from reading and deserializing a large number of KV tuples at once in the window operation with the AAR pattern.

In summary, Flink on FlowKV demonstrates the best performance in terms of tail latencies, while being solid without early failures.

6.3 Performance Breakdown

Figure 10 presents the CPU time breakdown of KV store operations when running the three queries with Flink. Overall, FlowKV spends significantly less (1.75×–10.56×) CPU time for store operations compared to RocksDB and Faster, indicating that FlowKV's semantic-aware designs actually lead to more efficient CPU use. In Q7 with the AAR pattern, data append and compaction are much more efficient in FlowKV thanks to the coarse-grained data organization. In Q11-Median with the AUR pattern, the predictive batch read contributes to efficient state read without frequent merging as in RocksDB, resulting in lower CPU usage in both data append and read. For Q11 with the RMW pattern, FlowKV demonstrates a more efficient update and read compared to Faster because FlowKV does not employ unnecessary synchronization mechanisms.

6.4 Sensitivity Analysis

We evaluate the effects of two crucial user-configurable parameters–predictive batch read ratio, and maximum space amplification (MSA)–on the performance of FlowKV.

Effect of Predictive Batch Read We experiment with the effect of predictive batch read using two queries with the AUR pattern, Q11-Median, and Q7-Session. Figure 11 shows the throughput changes and the hit ratio of prefetched tuples according to various read batch ratios. When the read batch ratio is zero (i.e., predictive batch read is disabled), the performance is, on average, only 38.3%-39.9% compared to that of Flink on FlowKV with predictive batch read, demonstrating the effectiveness of predictive batch read. Interestingly, Figure 11(a) shows that a read batch ratio higher than 0.02 does not actually bring performance benefits. This is because a high read batch ratio would also fetch KV tuples that have a low probability of being read as shown in Figure 11(b).

Effect of MSA (Maximum Space Amplification) For the AUR store, FlowKV performs compaction when the amplification caused by used data on the disk reaches a user-given threshold, MSA. MSA is defined as $N_{total}/(N_{total} - N_{outdated})$ where N_{total} is the number of files on the disk, and $N_{outdated}$ is the number of files that can be deleted. Compaction is called every time $N_{outdated}$ becomes too large so that the ratio exceeds the user-given MSA threshold.

Figure 12 explicitly shows the trade-off of two AUR queries. Smaller MSA saves disk space but calls compaction more frequently. On the other hand with larger MSA, disk space consumption increases, but fewer compaction calls are made. In our observation, we found that the performance tends to increase as the MSA increases. However, it did not show a significant difference after MSA of 1.5, which we think is the most conservative possible option without having to worry about disk space.

6.5 Multi-Machine Performance

Figure 13 demonstrates that Flink on FlowKV shows linear scalability to eight worker machines. This result is straightforward considering that FlowKV store instances are created for every physical window operation of streaming applications, posing no limitation on scalability.

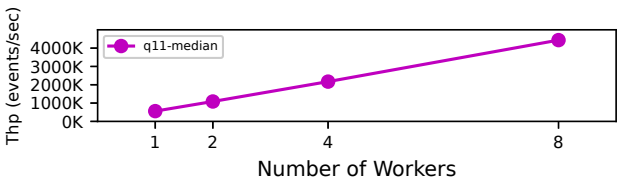

Figure 13. The maximum throughput of Q11-Median on the varying number of worker machines for window size 2000s.

7 Related Work

In our previous workshop paper [49], we investigated the performance problem of persistent KV stores that handle states of streaming applications. In this paper, we propose concrete approaches to tailoring data access by exploiting window semantics along with the complete design and implementation of FlowKV. We also perform an in-depth performance evaluation of FlowKV and other persistent KV stores deployed on Flink using the NEXMark benchmark.

Large State Management Efficiently managing the large-scale state of streaming applications has recently been an interest to research communities [1, 4, 29, 44, 49]. Earlier works [4, 44, 49] pointed out possible performance problems when managing internal states of streaming applications with persistent KV stores. Such findings motivated our development of FlowKV, a new semantic-aware composite store for stream processing.

Rhino [29] proposes a novel mechanism for efficient state migration and fault recovery for states whose sizes are larger than the main memory size. FlowKV, instead, focuses on efficient local state accesses for streaming applications, which makes it complementary to Rhino. Gadget [4] proposes a new benchmark for streaming state stores. While Gadget focuses on evaluating the independent performance of KV stores, our evaluation shows the end-to-end performance of streaming queries running on SPEs deployed on various KV stores. SummaryStore [1] optimizes storing and processing of colossal data streams by adopting approximation, which gradually decays the windowed data according to the elapsed time, which is orthogonal to FlowKV.

Persistent KV Stores Earlier works [8, 13, 32, 37, 51, 54, 56, 71, 72, 87] focused on building efficient KV stores working on persistent storage devices. Some design choices of FlowKV are inspired by these works, such as append-only logs [8, 32, 37], non-sorted data structures [13, 51, 87], and separation between metadata and actual data [56]. Since such KV stores are not aware of the semantics of window operations, they often become inefficient when handling states of streaming applications, as we explained in § 2.2.

Exploiting Modern Hardware for SPEs One of the recent trends of SPEs is tailoring stream processing on modern hardware, such as multi-core machines [59, 88], high-bandwidth memory [58], and FPGAs [81]. There have also been attempts to utilize heterogeneous coprocessors to synergistically process data streams on both CPU and GPU [16, 46, 60, 90]. FlowKV makes a new contribution by proposing a novel semantic-aware approach that efficiently utilizes modern persistent storage devices.

Window Optimization Efficient processing of window operations is an active research area. Research for optimizing a single and multiple sliding aggregation [7, 39, 48, 77, 83, 89] and window joins [30, 55, 63, 74, 78, 80, 86] have been proposed. FlowKV makes a new contribution to this area by designing and implementing a semantic-aware optimization for window state management on fast persistent storage.

Prefetching Compared to existing prefetching techniques [15, 31, 40, 41, 43, 50, 69, 91] that mainly use spatial or temporal locality, prefetching in FlowKV is specialized for streaming applications, exploiting semantics provided by these applications, such as information about *when* applications access data. The informed prefetching in FlowKV functions as a crucial design point that contributes to the good performance of FlowKV.

8 Discussion

Fault Tolerance Recovering internal states in case of failures is crucial for ensuring the integrity of streaming applications. Modern SPEs retrieve state changelogs from rewindable data sources (e.g., Kafka [47] and HDFS [76]), not from KV stores [12, 64]. In practice, persistency features of KV stores such as write-ahead logging are often disabled by SPEs for performance reasons [53]. In addition, SPEs periodically take snapshots of KV stores and store them into reliable storage such as HDFS [76] or S3 [3], which is called *checkpointing* [12, 64]. When a failure occurs, SPEs load the latest checkpoint and start replaying tuples from the checkpoint to minimize the time required for recovery. For this reason, FlowKV should be able to take a snapshot of its data, preferably in an asynchronous manner so that checkpointing does not block tuple processing. FlowKV can follow checkpointing strategies of other KV stores that are already deployed on SPEs. Flink, for example, forces the in-memory data of RocksDB to be flushed to disk before taking a snapshot, so that on-disk data can be transferred asynchronously while all the write operations are done in-memory [73].

Join Operations Join operations in stream processing merge two or more data streams and create a new joined data stream. As we have shown in § 6, FlowKV naturally supports windowed join operations (e.g., NEXMark Q8), which collect multiple streams in the same windows and perform joins on the bounded tuples inside each window. It would be an interesting direction to extend FlowKV to efficiently process other joins, namely, interval join operations [20].

Custom Window Operations When processing custom window operations (e.g., NEXMark Q4, Q6, and Q9), it is hard

to determine their state access patterns because their window functions are written in user code. FlowKV currently assumes that they have the unaligned read pattern (§ 3.1), but this may result in performance degradation because FlowKV might choose suboptimal stores and FlowKV cannot predict ETTs (estimated trigger times), which leads to frequent prefetch buffer misses. A common approach to address the problem is receiving hints from users [19, 24]. For example, the data access patterns of the custom window operations can be represented as annotations, such as `@UnalignedRead` or `@RMW`. FlowKV can also be provided with a user-defined function that calculates ETTs. Another interesting direction would be leveraging runtime profiling to determine optimal stores and ETTs. We leave this topic for future work.

Extending Existing KV Stores Existing KV stores can be extended in order to benefit from semantic-aware optimizations. Such extensions may provide new APIs that accept SPE information (e.g., timestamp and window assignment) to tailor data accesses. As an example, the extensions can predict future read candidates based on such information and prefetch candidate data by calling `multiget()`. They can also automatically configure the KV stores based on the access patterns of window operators [44]. For example, for RMW windows, RocksDB can get performance benefits by adopting hash write buffers with $O(1)$ access time.

9 Conclusion

FlowKV is a new semantic-aware persistent store specialized for stream processing engines. FlowKV exploits information about *how* and *when* window operations access data to tailor data accesses. Through our evaluation, we show that Flink on FlowKV has up to 4.12× higher throughput compared to Flink on RocksDB or Faster, while maintaining comparable tail latency. We also demonstrate that Flink on FlowKV scales out linearly on multiple machines.

Acknowledgments

We thank our shepherd Gala Yadgar and the anonymous reviewers for their insightful comments. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Nitin Agrawal and Ashish Vulimiri. 2017. Low-latency analytics on colossal data streams with SummaryStore. In *SOSP*.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB* 8, 12 (2015), 1792–1803.

[3] Amazon. 2022. Amazon S3. https://aws.amazon.com/s3/.

[4] Esmail Asyabi, Yuanli Wang, John Liagouris, Vasiliki Kalavri, and Azer Bestavros. 2022. A new benchmark harness for systematic and robust evaluation of streaming state stores. In *EuroSys*. 559–574.

[5] Showan Esmail Asyabi. 2021. Toward workload-aware state management in streaming systems. (2021).

[6] Lawrence Benson and Tilmann Rabl. 2022. Darwin: Scale-in stream processing. *CIDR* (2022).

[7] Pramod Bhatotia, Umut A Acar, Flavio P Junqueira, and Rodrigo Rodrigues. 2014. Slider: incremental sliding window analytics. In *Middleware*. 61–72.

[8] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better memory organization for LSM key-value stores. *VLDB* 11, 12 (2018), 1863–1875.

[9] Matthew Brookes. 2019. Moving on from RocksDB to something FASTER. In *Flink Forward*. Ververica.

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

[11] Bruno Cadonna and Dhruba Borthakur. 2021. How to tune RocksDB for uour Kafka Streams application. https://www.confluent.io/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/.

[12] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: Consistent stateful distributed stream processing. *VLDB* 10, 12 (2017), 1718–1729.

[13] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A concurrent key-value store with in-place updates. In *SIGMOD*. 275–290.

[14] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *USENIX FAST*. 17–32.

[15] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623.

[16] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. *HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines.* Technical Report.

[17] Apache Beam Community. 2016. Apache Beam. https://beam.apache.org/.

[18] Apache Beam Community. 2021. Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/.

[19] Apache Flink Community. 2022. Hints: Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/dev/table/sql/queries/hints/.

[20] Apache Flink Community. 2022. Joining: Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/dev/datastream/operators/joining/.

[21] Apache Flink Community. 2022. Windows: Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/dev/datastream/operators/windows/.

[22] Apache Kafka Community. 2017. State management. https://kafka.apache.org/23/documentation/streams/developer-guide/config-streams.

[23] Apache Samza Community. 2020. State management. https://samza.apache.org/learn/documentation/0.8/container/state-management.html.

[24] Apache Spark Community. 2022. Hints: Spark 3.3.0 documentation. https://spark.apache.org/docs/3.3.0/sql-ref-syntax-qry-select-hints.html.

[25] Apache Spark Community. 2022. Structured streaming programming guide. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[26] The Linux Community. 2022. dstat(1) - Linux man page. https://linux.die.net/man/1/dstat.

[27] The Linux Community. 2022. perf(1) - Linux man page. https://linux.die.net/man/1/perf.

[28] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *USENIX ATC*. 49–63.

[29] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *SIGMOD*. 2471–2486.

[30] Manuel Dossinger and Sebastian Michel. 2021. Optimizing multiple multi-way stream joins. In *ICDE*. 1985–1990.

[31] Sérgio Esteves, Joao Nuno Silva, and Luís Veiga. 2020. Palpatine: Mining frequent sequences for data prefetching in NoSQL distributed key-value stores. In *IEEE NCA*. 1–10.

[32] Facebook. 2012. RocksDB. http://rocksdb.org/.

[33] Facebook. 2021. Compaction. https://github.com/facebook/rocksdb/wiki/Compaction.

[34] Facebook. 2021. RocksDB tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[35] Fanrui. 2020. Optimization of Apache Flink for large-state scenarios. https://www.alibabacloud.com/blog/optimization-of-apache-flink-for-large-state-scenarios_597062.

[36] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A survey on the evolution of stream processing systems. *arXiv preprint arXiv:2008.00842* (2020).

[37] Google. 2011. LevelDB. https://github.com/google/leveldb.

[38] Brendan Gregg. 2017. Visualizing performance with flame graphs. (2017).

[39] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, and Alexandros Labrinidis. 2012. Three-level processing of multiple aggregate continuous queries. In *ICDE*. 929–940.

[40] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *MICRO*. 397–408.

[41] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G Abraham. 2004. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th annual international conference on Supercomputing*. 1–11.

[42] Cloudera Inc. 2020-2021. Configuring RocksDB state backend. https://docs.cloudera.com/csa/1.2.0/configuration/topics/csa-rocksdb-config.html?.

[43] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.

[44] Vasiliki Kalavri and John Liagouris. 2020. In support of workload-aware streaming state management. In *USENIX HotStorage*.

[45] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*. 783–798.

[46] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*. 555–569.

[47] J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *NetDB*.

[48] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *SIGMOD*. 623–634.

[49] Gyewon Lee, Jeongyoon Eo, Jangho Seo, Taegeon Um, and Byung-Gon Chun. 2018. High-performance stateful stream processing on solid-state drives. In *APSys*. 1–7.

[50] Sung Min Lee, Young Sun Youn, Su Kyung Yoon, and Shin Dug Kim. 2017. Intelligent clustering guided adaptive prefetching and buffer management for stream processing. In *IEEE SMC*. 2498–2503.

[51] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *SOSP*. 447–461.

[52] Shen Li, Paul Gerver, John MacMillan, Daniel Debrunner, William Marshall, and Kun-Lung Wu. 2018. Challenges and experiences in building an efficient apache beam runner for IBM streams. *VLDB* 11, 12 (2018), 1742–1754.

[53] Yu Li. 2020. GitHub: Apache Flink. https://github.com/apache/flink/blob/e85cf8c4cdf417b47f8d53bf3bb202f79e92b205/flink-state-backends/flink-statebackend-rocksdb/src/main/java/org/apache/flink/contrib/streaming/state/RocksDBResourceContainer.java.

[54] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*.

[55] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *SIGMOD*. 811–825.

[56] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM TOS* 13, 1 (2017), 5.

[57] Ovidiu-Cristian Marcu, Radu Tudoran, Bogdan Nicolae, Alexandru Costan, Gabriel Antoniu, and María S Pérez-Hernández. 2017. Exploring shared state in key-value store for window-based multi-pattern streaming analytics. In *IEEE CCGRID*. 1044–1052.

[58] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2019. StreamBox-HBM: Stream analytics on high bandwidth hybrid memory. In *ASPLOS*. 167–181.

[59] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2017. Streambox: Modern stream processing on a multicore machine. In *USENIX ATC*. 617–629.

[60] Georgios Michas, Periklis Chrysogelos, Ioannis Mytilinis, and Anastasia Ailamaki. 2021. Hardware-conscious sliding window aggregation on GPUs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–5.

[61] Microsoft. 2020. Tuning FasterKV. https://microsoft.github.io/FASTER/docs/fasterkv-tuning/.

[62] Microsoft. 2022. Structured streaming in production. https://docs.microsoft.com/en-us/azure/databricks/spark/latest/structured-streaming/production.

[63] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: A scalable, low-latency stream join architecture with adjustable ordering precision. In *USENIX ATC*. 493–505.

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

[65] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *USENIX ATC*. 537–550.

[66] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An efficient memory-mapped key-value store for flash storage. In *ACM SoCC*. 490–502.

[67] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing memory-mapped I/O for fast storage devices. In *USENIX ATC*. 813–827.

[68] Antonis Papaioannou and Kostas Magoutis. 2021. Amoeba: aligning stream processing operators with externally-managed state. In *UCC*. 1–10.

[69] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed prefetching and caching. In *SOSP*. 79–95.

[70] Jun Qin. 2021. Using RocksDB state backend in Apache Flink: When and how. https://flink.apache.org/2021/01/18/rocksdb.html.

[71] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*. 497–514.

[72] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *VLDB* 10, 13 (2017), 2037–2048.

[73] Stefan Richter and Chris Ward. 2018. Managing large state in Apache Flink: An intro to incremental checkpointing. https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html.

[74] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel index-based stream join on a multicore cpu. In *SIGMOD*. 2523–2537.

[75] Ning Shi and Seed Zeng. 2019. Stream processing with high cardinality and large state at Klaviyo. https://www.ververica.com/blog/stream-processing-with-high-cardinality-and-large-state-at-klaviyo.

[76] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. 2010. The hadoop distributed file system.. In *MSST*, Vol. 10. 1–10.

[77] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *VLDB* 8, 7 (2015), 702–713.

[78] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos Mamoulis. 2005. RPJ: Producing fast join results on streams through rate-based optimization. In *SIGMOD*. 371–382.

[79] Micron Technology. 2013. SSDs for big data – Fast processing requires high-performance storage. http://docs.media.bitpipe.com/io_12x/io_127132/item_1233643/brief_ssds_big_data.pdf.

[80] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *SIGMOD*. 625–636.

[81] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A framework for massively parallel streaming on FPGAs. In *ASPLOS*. 639–651.

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

[83] Jonas Traub, Philipp Marian Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2021. Scotty: General and efficient open-source window aggregation for stream processing systems. *ACM TODS* 46, 1 (2021), 1–46.

[84] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *Nexmark–a benchmark for queries over data streams.* Technical Report. OGI School of Science & Engineering at OHSU.

[85] Stefan van Wouw and Max Thone. 2020. Performant streaming in production: Preventing common pitfalls when productionizing streaming jobs. https://databricks.com/session_na20/performant-streaming-in-production-preventing-common-pitfalls-when-productionizing-streaming-jobs.

[86] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. *VLDB* 29 (2003), 285–296.

[87] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *USENIX ATC*. 71–82.

[88] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *VLDB* 12, 5 (2019), 516–530.

[89] Chao Zhang, Reza Akbarinia, and Farouk Toumani. 2021. Efficient incremental computation of aggregations over sliding windows. In *ACM SIGKDD*. 2136–2144.

[90] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures. In *USENIX ATC*. 633–647.

[91] PengFei Zhu, GuangYu Sun, Peng Wang, and MingYu Chen. 2015. Improving memory access performance of in-memory key-value store using data prefetching techniques. In *International Workshop on Advanced Parallel Processing Technologies*. Springer, 1–17.