Exploring Shared State in Key-Value Store for Window-Based Multi-Pattern Streaming Analytics

Ovidiu-Cristian Marcu, Radu Tudoran, Bogdan Nicolae, Alexandru Costan, Gabriel Antoniu, María S. Pérez-Hernández

Exploring Shared State in Key-Value Store for Window-Based Multi-Pattern Streaming Analytics

Ovidiu-Cristian Marcu*, Radu Tudoran†, Bogdan Nicolae†,
Alexandru Costan*, Gabriel Antoniu*, María S. Pérez-Hernández‡

*IRISA/INRIA Rennes Bretagne Atlantique
{ovidiu-cristian.marcu, alexandru.costan, gabriel.antoniu}@inria.fr
†Huawei Research Germany
{radu.tudoran, bogdan.nicolae}@huawei.com
‡Universidad Politecnica de Madrid
mperez@fi.upm.es

***Abstract*—We are now witnessing an unprecedented growth of data that needs to be processed at always increasing rates in order to extract valuable insights. Big Data streaming analytics tools have been developed to cope with the online dimension of data processing: they enable real-time handling of live data sources by means of stateful aggregations (operators). Current state-of-art frameworks (e.g. Apache Flink [1]) enable each operator to work in isolation by creating data copies, at the expense of increased memory utilization. In this paper, we explore the feasibility of deduplication techniques to address the challenge of reducing memory footprint for window-based stream processing without significant impact on performance. We design a deduplication method specifically for window-based operators that rely on key-value stores to hold a shared state. We experiment with a synthetically generated workload while considering several deduplication scenarios and based on the results, we identify several potential areas of improvement. Our key finding is that more fine-grained interactions between streaming engines and (key-value) stores need to be designed in order to better respond to scenarios that have to overcome memory scarcity.**

***Index Terms*—Big Data, memory deduplication, streaming analytics, sliding-window aggregations, Apache Flink.**

1. Introduction

Data is the new natural resource. Its ingestion and processing is nowadays transformative in all aspects of our world. However, unlike natural resources, whose value is proportional to the scarcity, the value of data grows larger the more of it is available. This trend is facilitated by big data analytics: more data means more opportunities to discover new correlations and patterns, which leads to valuable insight.

Unsurprisingly, data is accumulating at fast rates: predictions show it will reach the order of Zettabytes by 2020. As a consequence, big data analytics techniques used to process the data face major challenges in terms of scalability, performance and resource efficiency. In this context, live data sources (e.g., web services, social and news feeds, sensors, etc.) are increasingly playing a critical role in big data analytics for two reasons: first, they introduce an online dimension to data processing, improving the reactivity and "freshness" of the results, which can potentially lead to better insights. Second, processing live data sources can offer a potential solution to deal with the explosion of data sizes, as the data is filtered and aggregated before it gets a chance to accumulate. Thus, stream-oriented data processing engines specifically designed to ingest and operate on continuous (unbounded) data streams (such as *Storm* [2] and *Flink* [1]) saw a rapid rise in popularity.

Stream-oriented engines typically process live data sources using stateful aggregations (called operators) defined by the application, which form a directed acyclic graph through which the data flows. In this context, it is often the case that such stateful aggregations need to operate on the same data (e.g. top-K and bottom-K entries observed during the last hour in a stream of integers). Current state-of-art approaches create data copies that enable each operator to work in isolation, at the expense of increased memory utilization. However, with increasing number of cores and decreasing memory available per core [5], memory becomes a scarce resource and can potentially create efficiency bottlenecks (e.g. underutilized cores), extra cost (e.g. more expensive infrastructure) or even raise the question of feasibility (e.g. running out of memory). Thus, the problem of minimizing memory utilization without significant impact on the performance (typically measured as result latency) is crucial.

In this paper, we explore the feasibility of deduplication techniques to address this challenge. What makes this context particularly difficult is the complex interaction and concurrency introduced by the operators as they compete for the same data, which is not originally present in the case when operators work in isolation. We summarize our contributions as follows:

- We formulate the problem of deduplication in the context of stream processing (Section 2).
- We design a deduplication technique specifically for window-based operators that relies on key-value stores to hold a shared state. We illustrate the implementation of this technique using a production-ready stream processing engine: Apache Flink (Sections 3, 4.3).
- We experiment with a synthetically generated workload in several deduplication scenarios and setups. In particular, we study the latency under weak and strong scalability using two different key-value stores and comment on the corresponding memory utilization (Section 5).
- Based on the results, we identify several potential areas of improvement and comment on the associated research opportunities (Section 7).

2. Background

This section discusses the general context of this work, targeted use cases and the main working assumptions. Based on this we introduce the problem statement.

2.1. Context

Streaming becomes a key processing paradigm driven by the need of many applications and scenarios to react fast to continuously arriving events. The increase demand for fast and smart decisions is not specific to a single domain. Whether we discuss IoT (e.g., smart manufacturing, smart factories), finance, autonomous driving, smart spaces or smart cities, gaming, ecommerce; applications share the need of running analysis against each incoming event and generating results with low latencies. Even if the analysis can vary in scope across such domains, typical streaming patterns of data processing are filtering, projecting, data structure (i.e., event) enhancements, aggregates and custom UDF (user defined functions). One can observe the trend of such computation in the semantics of streaming APIs and the efforts for unifying the streaming semantics across engines ([7], [15], [21]).

Stream processing window functions such as aggregates and UDF (i.e., **patterns**) are more challenging as they pre-require buffering the data over some periods of times (i.e., these functions are typically applied over the window contents). The functions that are applied are quite generic and range from mathematical functions (e.g., computing statistics, histograms) to extracting data features for machine learning or for business intelligence (e.g., min, max, summations, metrics over partitions) to binary or multivariate functions (e.g., labeling items as relevant or irrelevant in a specific context). To exemplify, one can consider the example of gaming specific scenarios [3], which puts in evidence Terabytes of state generated by billions of events per day. The processing focuses on computing revenue streams in real time (e.g., summations - total revenue, metrics over partitions - computing average revenues per country) and on determining user activities (i.e., labeling functions - which levels make user quit; histograms - hourly activities for games), etc.

Multi-Patterns. This paper considers the general case of applying such aggregations and UDFs (two or more patterns) over partial or full common stream data, and without focusing on a particular domain. We consider how the underlying stream operator (i.e., the window) can better support these concurrent analysis and make resource usage more efficient (e.g., decrease memory footprint) without leveraging properties (e.g., associativity) of the patterns' functions that are applied. This raises additional challenges with the use cases where no specific assumptions can be made, other than the ones that are generally considered by the stream paradigm; on the other hand the approaches considered need to be transparently encapsulated within the stream framework without altering the stream paradigm or the API semantics.

2.2. Problem Statement

We define the working scenario as follows: we have a rate of new events (typically few thousand events per second - half a billion events per day). This is a general assumption on the event workload that applies across the aforementioned domains: IoT, banks, gaming companies, e-commerce sites have events in the range of million to tens of millions per day (e.g., a large game company will have about 30 million events per day). We consider analysis history up to 12 months of historical events. This can cover analysis from instant metrics to complex machine learning algorithms that aim to learn user behavior, which require large time-spans. In terms of domain parallelism, we build millions of windows (each event can be associated to one or multiple windows) that we keep as state in memory in order to process *multiple patterns* (that correspond to window-based UDF or aggregations). The choice for this granularity is motivated by the fact that banking or ecommerce have millions of users. Furthermore, the specific analysis can require various partitions (e.g., computing averages per user, per country or per currency) which drive the need to associate each event with multiple windows to support the corresponding processing. Each event value size is *significant* (hundreds of bytes) and correspond to multiple attributes that are possibly used in each pattern's computations. The arity of the tuples can range from tens (e.g., data specific to financial markets) to hundred attributes (data in e-commerce is large and augmented with metadata from various cookies).

The computation will thus contain multiple window processing operators (N) that are running concurrently within the stream engine, in order to process windows built from the same input of infinite events. In Listing 1 (Flink's API) we give an example of building a topology with two patterns running window functions on the same data stream: after creating an input *DataStream* by parsing events from one source (*readParseSource*), we subsequently define two patterns as window operators. Current implementations are based on duplicating stream events in memory, leading to inefficient memory usage and potentially increased processing event latency. Consequently, the memory footprint is equal to the sum of the states of all processed windows. One can imagine that if the number of pattern analysis that run in parallel grows, we can end up with a several tenfolded multiplication factors over the entire data.

The goal of this paper is to explore the possibility to store the shared state in an external key-value store in order to efficiently deduplicate memory corresponding to events that are common to multiple (overlapping) window-based operators.

Listing 1: Two patterns on common data stream

```
DataStream<EventType> input = env.readParseSource(params);

DataStream<ResultType1> patternOne = input
.keyBy(<first key selector>)
.window(<first window assigner>)
.<window transformation>(<first window function>);

DataStream<ResultType2> patternTwo = input
.keyBy(<second key selector>)
.window(<second window assigner>)
.<window transformation>(<second window function>);
```

3. Memory Deduplication with Shared State Backend

In this section, we briefly introduce the concept of stateful window-based stream processing and propose a deduplication approach specifically designed for this context.

3.1. Stateful Window-Based Processing

At its basis, an infinite data stream is a set of events or tuples that grows indefinitely in time [16]. An infinite data stream is divided (based on event timestamp or other attributes) into finite slices called windows [4]. The properties of a window are determined by a window assigner: it specifies how the elements of the stream are divided into windows. The main categories are:

- *global windows*: each element is assigned to one single per-key global window;
- *tumbling windows*: elements are assigned to fixed length, non-overlapping windows of a specified window size;
- *sliding windows*: elements are assigned to overlapping windows of fixed length equal to the window size, the size of the overlap is defined by the window slide; and
- *session windows*: windows are defined by features of the data themselves and window boundaries are adjusting to incoming data.

Stateful operators implemented as (sliding) window-based aggregations are working over a state that defines the confines of the (sliding) window. The window state is a set of *M* recent tuples and is usually persisted as a list structure in heap memory or off-heap embedded key-value store. The implementation can also be hybrid, with references (hash keys) of tuples stored in heap memory and actual values stored in an external key-value store.

To build and modify a window state, the (evicting) window operator is using a *ListState* interface that gives access to various methods to add a tuple to the state , remove a tuple from the state or retrieve all the tuples of the state. *ListState* methods can be defined for both generic tuples and serialized (byte array) ones, depending on the method used to persist state in memory (storing tuples in a serialized format helps reduce the memory footprint with increased cpu usage).

The window state backend abstraction is hidden from the developer, but can be parametrized in order to use different implementations.

3.2. Deduplication Proposal

Before we try to find an efficient way of reducing the pressure on memory for persisting window states, it is important to understand what properties of user-defined functions can lead to a reduction of the state and thus reduced memory utilization.

As discussed in [20], if the aggregation function is associative (not necessary to be commutative or invertible), then a general *incremental* approach could possibly avoid buffering window states. It can help to achieve much better event latency for large windows, while the memory footprint for storing partial aggregates is much lower than in the case of storing entire windows. For small windows, it provides almost the same event latency.

However, in some cases, there is a need to access the elements of a window after the aggregation was executed, so although incremental aggregation can be efficient, so a window state may still be necessary. If we consider that not all the aggregation functions are associative, than we are forced to re-aggregate from scratch for each window update.

Our approach for window states memory deduplication is based on the following: for each element (event value) of a stream we calculate and associate a key (reference). Each window's buffer is defined as a list of references to the assigned events as follows:

```
WindowKey -> ListStruct<EventReference>
```

Based on the properties of the windows (how elements are arriving, ordering, eviction policies), *ListStruct* may be implemented as a simple list or as a more complex structure.

Each event value with associated reference will be stored once in a key-value store and accessed every time a window aggregation is activated.

Existing approaches do not consider sharing a window's state elements. The analyzed framework (Apache Flink) is caching buffers in either JVM heap (leading to increased memory footprint because of Java representation overhead) or to an embedded key-value store (RocksDB), possibly wasting memory resources because of duplicated stream events. As such, our approach is worth being explored in order to respond to critical situations where memory usage needs to be reduced.

4. State Backend Options for Window-Based Processing

In this section we describe the current possibilities to work with window-based state backends in Apache Flink and we analyze a set of optimizations. Next, we describe the implementation of the proposed shared-state backend and we detail the necessary enhancements added to Flink's interfaces.

Apache Flink gives three ways of storing state for window-based operators:

1) Memory state backend: it stores its data in heap memory with no capabilities to spill to disk;
2) File state backend: it stores its data in heap memory and it is backed by a file system;
3) Embedded key-value store (RocksDB) state backend: it stores its data in RocksDB with capabilities to spill to disk.

We choose Apache Flink [15] to develop a proof of concept of our techniques, as it is today the most advanced open-source streaming engine. Flink adopts most of the Dataflow window model as described in [14], being the state of the art windowing semantics.

Let us now discuss the options we can consider for window state (buffering) backends. While some of the options are currently implemented (heap and rocksdb), some other states are proposed by us and used in our evaluation (heap+redis, rocksdb+redis).

4.1. Objects State in Heap

By default Flink stores data internally as objects on the Java heap in a memory state backend which has strong limitations: 1) the size of each individual state is limited to a few megabytes; 2) the aggregate state must fit into the configured job heap memory.

In our window-based scenarios (i.e. jobs with large state, many large windows) we are required to save each window operator's instance states on the local task manager heap memory. For this situation we can configure Flink to a file state backend which is characterized by holding data in the task manager heap memory and further checkpointing the state into a file system (e.g. HDFS) in order to ensure consistency guarantees. To configure this state we have to initialize two parameters: 1) *state.backend* to value *filesystem* and 2) *state.backend.fs.checkpointdir* to the HDFS path for checkpointing state. Each operator window's state is a list of Java objects and it is updated every time a new element arrives.

4.2. Serialized Objects State in Off-Heap

Similar to the heap object state, Flink offers an option to configure an operator state to off-heap and it implements an embedded key-value store state interface (i.e. RocksDB). The main difference is that objects are serialized before they are persisted in the off-heap state and every time objects are accessed the cost of deserialization adds to the processing latency of corresponding operator's user defined function. Another difference consists in the fact that the RocksDB database is using local task manager data directories and as such the state size is limited by the amount of disk space available.

4.3. Memory Deduplication with Shared Key-Value Store

Let us now describe our proposed solution for storing shared state for memory deduplication purposes. For each new object that is assigned to an operator's window, we calculate a key by hashing the value of the event. We implement a new interface *SharedListState* that is configurable by setting the parameter *state.backend* to *sharedfilesystem*. When we add a value to the shared state we make the following operations: 1) we append the reference key to a list and we store this list in JVM heap memory; 2) we store the *key, serialized value* pair in the external key-value store. When an operator's window execution is triggered because a new event arrived, we retrieve the list of keys from heap in order to make a call (multi-get) to the external key-value store in order to obtain all the serialized values. We subsequently deserialize each value and further trigger the user defined function that computes the window aggregation. Our approach is not only effective for memory deduplication, but will also be useful for moving computation to other nodes (separating state from the streaming execution) if we consider that our key-value store is configured to replicate its data.

5. Experimental evaluation

This section describes the experimental setup, methodology and results.

5.1. Setup

We implemented an event generator that is capable of streaming events through a socket. As a motivating scenario, the event generator is designed to emulate user transactions in a banking system, which are used in a fraud detection scenario. Specifically, the user transactions are strings (events) composed of relevant attributes (type of transaction, date, merchant name, value of transaction, type of card, name of customer). Their content is generated randomly, according to the following distribution (in order to draw one real scenario where events arrives uniformly): an equal number of twelve events in a number of steps proportional to 1000 milliseconds (e.g., 60 events are streamed as 12 events every 200 milliseconds).

The user transactions are consumed by a Flink application that operates on some of the parameters through a user-defined aggregation operator. The operators we implemented perform two low-cpu metrics (sum, min).

We deploy Flink standalone (version 1.1 modified with our shared state backend approach) on a single node which has an Intel Xeon CPU E5-2630 v3 @ 2.40GHZ X 16, Ubuntu 16.04 LTS 64-bit, 31 GB RAM and 512 GB disk.

5.2. Methodology

For every experiment we follow a similar cycle. We install Redis 3.2.4 [6] and we configured a standalone Flink to use it as the state backend. We start the event generator as a Java socket program that listens to a configured port. First, it generates strings (events) until they fill the window state. Then, it generates strings for five iterations, each of one minute, keeping the same rate of new events. At the same time, the Flink application uses the *socketTextStream* method of the *StreamExecutionEnvironment* in order to create a new data stream that contains the strings received from the configured socket. We parse each event with a *flatMap* operation, assigning it a timestamp. After applying a user-defined function (as mentioned in the previous section), the timestamp is used to obtain the event latency, defined as the time seen at the end of the aggregation minus the initial time of the last event of a window. We collect each aggregated value and write it as text with the operator *writeAsText*.

We measure only the event latencies corresponding to the five iterations and we compute the aggregated upper bound of latencies experienced by 99% of events.

We make sure to clear the OS buffer cache and temporary data or logs before a new execution starts. After each execution ends we clear the Redis cache and we collect logs that hold the event latency percentiles.

For each experiment we fix the values for the following parameters (while keeping all other parameters at their default): (1) *window_size* – is the size of the window for which we execute an user defined function aggregation, window slide is fixed to 1 (in order to put pressure on window evaluation); (2) *window_keys* – gives the number of windows that are processed in parallel, equals the number of cores; (3) *events_rate* – is the rate of new events that are streamed by the socket program each second according to the distribution mentioned in the previous section. Other parameters: (4) size of event reference (key) is 16 bytes (to avoid colisions for one year worth of data); (5) size of each event value is constant 100 bytes (estimated size for a typical user transaction).

5.3. Impact of Heap Size on Event Latency

Our first series of experiments aims at understanding the overhead of operating under low memory constraints, which leads to frequent invocation of the garbage collector and thus decreased performance. To this end, we use a variable heap memory size for the task manager keeping the other parameters constant. We choose to evaluate a low cpu aggregation operator over a sliding window of size *10240* (slide equal one) with the rate of new events set at *480* events per second. The number of processed windows is two and we only use one task slot (parallelism one). The window state is configured to use the heap.

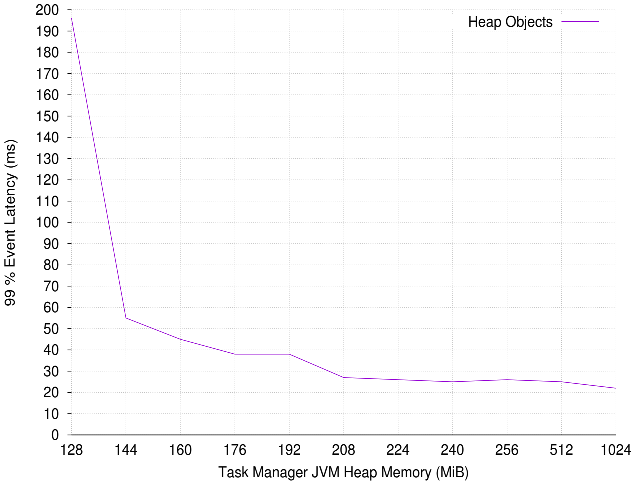

Figure 1: Event processing latency 99% percentile for fixed window size and event rate when variating Task Manager Heap Size.

We observe that the cpu usage decreases as a consequence of a reduced garbage collector overhead, which leads to a decrease in measured event latency of up to ten times (as observed in Figure 1). This emphasizes the importance of avoiding running stream processing operators under low memory constraints.

5.4. Impact of Window Size on Event Latency

Next, we evaluate the impact of the window size on the perceived event latency, which is the main indicator of performance in a stream processing application.

In Figure 2 we observe that with larger windows the effect of queueing on the event latency increases. Specifically, the event latency is composed of the following breakdown: event queueing (how much time an event is buffered before it got its chance to be processed), overhead to add an event to state, overhead to retrieve the whole window from the state, time to process aggregation, framework overhead.

We also evaluate the off-heap object serialization option using *RocksDB* as the state backend (assuming default configuration in Flink of the embedded key-value store) for the same rate of 60 new events per second. We do not plot these numbers as they show much higher latencies: 111 milliseconds for windows of size 1024, 310 milliseconds for windows of size 2048 and hundreds of seconds for a window of 4096 events.

To facilitate a feasible comparison between “redis-dedup” using rocksdb and plain rocksdb, we decreased the rate of new events to 36 and we also reduced the window

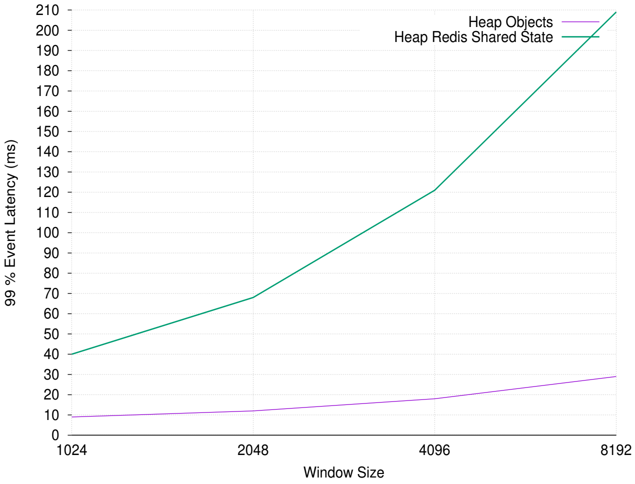

Figure 2: Event processing latency 99% percentile for fixed event rate when using a variable window size. Rate of new events is 60. Heap size is 1GB. Parallelism is one: all events correspond to the same window.

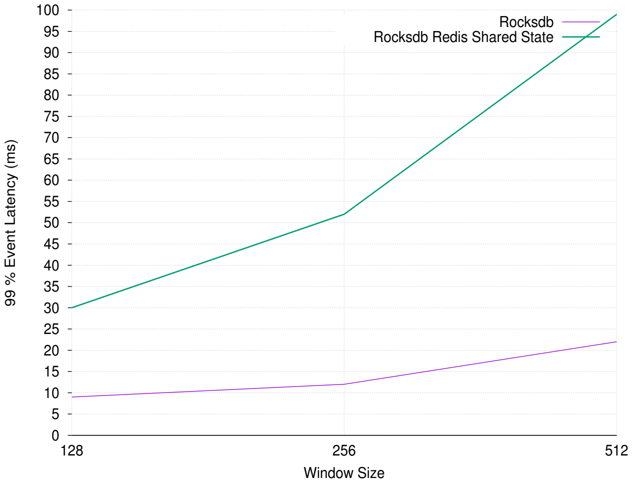

Figure 3: Event processing latency 99% percentile for fixed event rate when variable window size. Rate of new events is 36. Heap size is 1GB. Parallelism is one: all events correspond to the same window. Heap event latency is 5 milliseconds.

size. The results in Figure 3 show that although the size of the windows is much smaller compared with the previous experiment, the overhead of using RocksDB as a state backend is much higher. Nevertheless, the effect of event queueing follows a similar trend.

5.5. One versus Two Operators using Shared Events

The next experiment evaluates the event processing latency when increasing the number of operators sharing the same events from one to two. To this end, we fix the problem size per core as follows: for each core we generate 12 events every 200 milliseconds, considering that events are

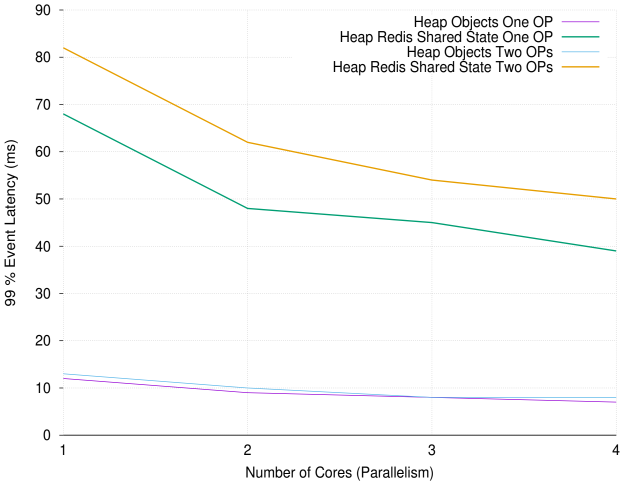

Figure 4: Event processing latency 99% percentile for fixed event rate when variating parallelism. Rate of new events is 60 per core every second. Heap size is 1GB. Window size is 2048.

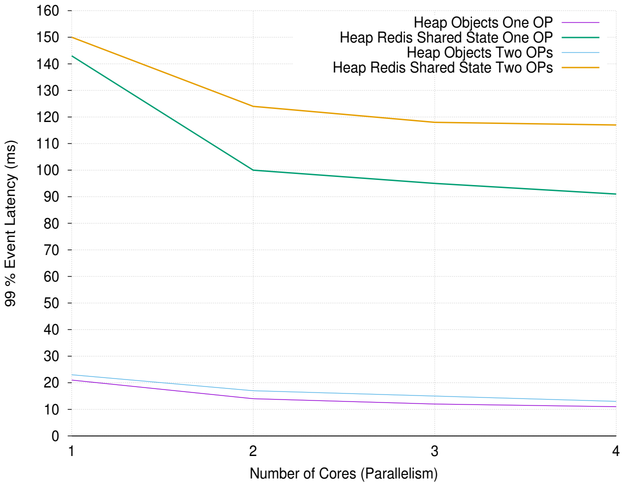

Figure 5: Event processing latency 99% percentile for fixed event rate when variating parallelism. Rate of new events is 120 every second. Heap size is 1GB. Window size is 4096.

all part of the same window. We evaluate up to four windows corresponding to the same parallelism of each operator.

In Figure 4 we plot four benchmarks: two correspond to the application of one operator having state in heap (Heap Objects One OP) or sharing state in Redis (Heap Redis Shared State One OP) and the other two correspond to the application of two operators having state in heap (Heap Objects Two OPs) or sharing state in Redis (Heap Redis Shared State Two OPs).

While for heap state we do not have differences between observed latencies, for shared state we observe an almost constant gap between each execution. This gap is related to the increased pressure on the key-value store, trying to access the same data concurrently.

As we can see in Figure 5, this experiment evaluates the same problem (events rate of 120 per second, window size of 4096 events) while increasing the parallelism from one to four (each core will process an equal number of elements) for the same four benchmarks like in the previous section.

It is clear that with larger windows and increased throughput, relying on an external key-value store for memory deduplication of window-based processing may be unfeasible due to large overheads of serialization.

5.6. Memory Savings

Scenario	Experimental Use Case	Large Use Case
Heap Only	800 KB	4.7 TB
Deduplicated	592 KB	1.3 TB

TABLE 1: Estimating the memory utilization for the use case presented in the previous section (N is 2, M is 4096) and a potential large use case at scale (N is 10, M is one month worth of 2000 events per second) with and without deduplication.

Although there is significant performance overhead when using deduplication (as detailed in the previous section), such a technique may lead to large memory savings. While we did not measure the memory utilization directly (which is inherently difficult due to garbage collection), we estimate it as $M * N * EventSize$ for the heap-only case and, respectively $M * EventSize + (N + 1) * M * KeySize$ for the deduplicated case.

In these formulas, N is the total number of concurrent operators, M is the total number of events, $EventSize$ is fixed at 100 bytes, $KeySize$ is fixed at 16 bytes. All operators are assumed to consume the same events (i.e. full window overlap).

The results of our estimation, both for the use case used in our experiments as well as a larger hypothetical use case at scale are summarized in Table 1. For the case where only two operators share common data, it can be observed that deduplication leads to a consistent memory saving of 26% compared with the heap-only case, while for a large scale use case with ten operations sharing data we estimate that deduplication leads to higher memory savings of up to 72% compared with the heap-only case.

6. Related Work

Deduplication is a common technique used in a variety of scenarios, both obvious (e.g., saving space in file systems [22], [11] or reducing the size of large scale memory dumps [18]) and less obvious (e.g. detection of natural replicas to reduce the cost of replication-based resilience [19]).

In the context of stream computing, recent research efforts have concentrated on the problem of sharing the state for overlapping sliding windows over event streams. However, as described below, they focus on very specific issues, which they alleviate in isolation, in most cases trading performance for expressivity.

Exploiting data redundancy. In [17], the authors introduce a buffer management algorithm that exploits the access pattern of sliding windows in order to efficiently handle memory shortages. The idea is that sliding-window operators are most of the time manipulating only a small fraction of their data set and are doing so in a very predictable pattern: once a tuple is stored on the window it is not going to be accessed by the sliding-window operator until it is time to expire it. Hence, they implement a shared operator working over multiple windows, consisting of tuples in a shared tuple repository.

However, they only consider time-based windows and it is not clear how tuples are referenced from a window. Also, the total space of the shared repository is the largest window. This means that the algorithm only works for the particular case of windows that overlap in a deterministic way (with a coarse granularity of one hour), all included in a larger, containing window. In contrast, our approach also supports *fragmented intersections of windows* and that with a *finer granularity* (i.e. an event).

Tuples referencing. The idea of using pointers to the data tuples was introduced in the Continuous Query Language (CQL) [8], an SQL-based declarative language for registering continuous queries against streams and updatable relations. In CQL windows are implemented by non-shared arrays of pointers to shared data items, such that a single data item might be pointed to from multiple windows. To minimize copying and proliferation of tuples, all tuple data is stored in synopses (i.e. in-memory hash tables) and is not replicated. Synopses are used at runtime to compile the query plans, which are merged whenever possible, in order to share computation and state.

Similarly to us, CQL uses the same idea of pointers to shared data items, yet the effects of serialization/deserialization as well as the heap buffer limitations are not assessed. Deduplication only works as a side-effect of merging various query plans. Queues contain references to tuple data within synopses, along with tags containing a timestamp and an insertion/deletion indicator. Our contribution is the *generalization of synopses into shared state* as key-value stores.

Incremental event processing. Sliding-window aggregation is a key operation in stream processing and incremental aggregations help to avoid re-aggregating from scratch after each window change. In order to exploit this incremental processing, the targeted functions need all to be associative (e.g. count, sum, max, min, mean etc.). Without the associativity, one can only handle insertions one element at a time at the end of the window. Hence, associativity enables breaking down the computation in flexible ways. Reactive Aggregator [20] is an example of such a framework for incremental sliding-window aggregation. It stays competitive (10% higher throughput than from-scratch re-computation) on small windows (1 to 100 events), but its true performance is shown for large windows (thousands of events), when it delivers at least one order of magnitude higher throughput compared to re-executions. Orthogonal to Reactive Aggregator is Cutty [10], a project that is considering more general non-periodic windows (punctuations and sessions, custom deterministic windows) which are expressed as user-defined operators. Cutty relies on a technique to discretize a stream into a minimal set of slices for efficient aggregate sharing of user-defined windows. Shared data fragments [12] focus on various techniques for aggregation sharing without the need of optimizing queries upfront. Our work is orthogonal to these solutions considering the more general user defined functions that are not associative.

Aggregate computation optimizations. In [13] the authors introduce the panes technique to evaluate sliding-window queries by pre-/sub-aggregating and sharing computations. The problem of efficiently computing a large number of sliding-window aggregates over continuous data streams was introduced for the first time in [9]. The authors put forward three cost parameters that need to be considered: a) memory required to maintain state (space); b) the time to compute an answer (lookup time) is proportional to the window size; c) the time to update the window state when a new tuple arrives (update time) is composed of the time to evict old tuples as well. While their focus is on pre-aggregating values that are eventually shared, our approach maintains full window-state in memory. As opposed to the case of associative functions, for which incremental processing was a good option, this is no longer the case in the general (non-associative) scenario. In this context, our aggregation functions are able to reconsider all tuples of a window for each evaluation and to gracefully compute from scatch.

7. Conclusions

Stream processing is a key big data analytics technology that enables extracting insight in a timely fashion while avoiding excessive data accumulation by means of states. However, current state-of-art approaches enable each operator to work in isolation by creating data copies, at the expense of increased memory utilization. Given the scarcity of memory due to increasing application complexity and decreasing memory per core (a trend of modern multi-core architectures), the problem of optimizing memory utilization for stream processing becomes critical. This paper contributes with a study on the feasibility of deduplicating the shared data across the operator states through a technique illustrated using Apache Flink [1].

Based on this study, we draw three conclusions. First, under low memory constraints, operators tend to perform poorly due to frequent invocations of the garbage collector. In this case, the latency needed to process 99% of the events is up to 10x higher compared with the case when there is no memory pressure. Thus, deduplication has potential to improve latency. However, deduplication leads to higher performance degradation for an increasing window size compared to the case when copies are used. Thus, careful selection of the window size is needed. Third, deduplication has a large potential to save memory: already for two operators that share the same state there is a 25% reduction, which continues to grow proportionally with the number of operators sharing the same state.

Encouraged by these results, we plan to explore deduplication in greater depth in future work. First, we aim to extend the evaluation at large scale in multi-node scenarios where deduplication is performed also between nodes. Second, we plan to explore multi-pattern aggregations using key-value stores that support group queries. Finally, we also plan to explore other streaming-oriented optimizations like UDF aggregations and arrays instead of individual tuples.

Acknowledgment

This work is part of the BigStorage project, funded by the European Union under the Marie Sklodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). It is also supported by the ANR OverFlow project (ANR-15-CE25-0003).

References

[1] Apache Flink. http://flink.apache.org.

[2] Apache Storm. http://storm.apache.org.

[3] Flink large state use case. http://flink-forward.org/wp-content/uploads/2016/07/Gyulo-Fo%CC%81ra-RBEA-Scalable-Real-Time-Analytics-at-King.compressed.pdf.

[4] Flink windows. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html.

[5] Hardware trends in keynote. http://www.pdsw.org/keynote.shtml.

[6] Redis. https://redis.io/download.

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

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[9] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 336–347. VLDB Endowment, 2004.

[10] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1201–1210, New York, NY, USA, 2016. ACM.

[11] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: a scalable secondary storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 197–210, San Francisco, USA, 2009. USENIX Association.

[12] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 623–634, New York, NY, USA, 2006. ACM.

[13] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, March 2005.

[14] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 311–322, New York, NY, USA, 2005. ACM.

[15] Björn Lohrmann, Daniel Warneke, and Odej Kao. Nephele streaming: Stream processing under qos constraints at scale. *Cluster Computing*, 17(1):61–78, March 2014.

[16] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of data streams and operators. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 37–52, Berlin, Heidelberg, 2005. Springer-Verlag.

[17] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. Overcoming memory limitations in high-throughput event-based applications. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 399–410, New York, NY, USA, 2013. ACM.

[18] Bogdan Nicolae. Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. In *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, pages 19–28, Boston, USA, 2013.

[19] Bogdan Nicolae. Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead. In *IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium*, pages 1023–1032, Hyderabad, India, 2015.

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

[21] Matei Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM SOSP*, pages 423–438, NY, USA, 2013. ACM.

[22] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 18:1–18:14, San Jose, USA, 2008. USENIX Association.