Challenges and Experiences in Building an Efficient Apache Beam Runner For IBM Streams

Shen Li† Paul Gerver* John MacMillan*
Daniel Debrunner* William Marshall* Kun-Lung Wu†

{shenli, pgerver}@us.ibm.com, johnmac@ca.ibm.com,
{debrunne, wcmarsha, klwu}@us.ibm.com

†IBM Research AI *IBM Watson Cloud Platform

ABSTRACT

This paper describes the challenges and experiences in the development of IBM Streams runner for Apache Beam. Apache Beam is emerging as a common stream programming interface for multiple computing engines. Each participating engine implements a *runner* to translate Beam applications into engine-specific programs. Hence, applications written with the Beam SDK can be executed on different underlying stream computing engines, with negligible migration penalty. IBM Streams is a widely-used enterprise streaming platform. It has a rich set of connectors and toolkits for easy integration of streaming applications with other enterprise applications. It also supports a broad range of programming language interfaces, including Java, C++, Python, Stream Processing Language (SPL) and Apache Beam. This paper focuses on our solutions to efficiently support the Beam programming abstractions in IBM Streams runner. Beam organizes data into discrete event time windows. This design, on the one hand, supports out-of-order data arrivals, but on the other hand, forces runners to maintain more states, which leads to higher space and computation overhead. IBM Streams runner mitigates this problem by efficiently indexing inter-dependent states, garbage-collecting stale keys, and enforcing bundle sizes. We also share performance concerns in Beam that could potentially impact applications. Evaluations show that IBM Streams runner outperforms Flink runner and Spark runner in most scenarios when running the Beam NEXMark benchmarks. IBM Streams runner is available for download from IBM Cloud Streaming Analytics service console.

PVLDB Reference Format:
Shen Li, Paul Gerver, John MacMillan, Daniel Debrunner, William Marshall, Kun-Lung Wu. Challenges and Experiences in Building an Efficient Apache Beam Runner For IBM Streams. *PVLDB*, 11 (12): 1742-1754, 2018.
DOI: https://doi.org/10.14778/3229863.3229864

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229864

1. INTRODUCTION

Distributed stream computing has been adopted by a rapidly increasing number of industrial applications. For example, social networks use streaming analytic systems to detect trending topics and react to news events. Peer-to-peer ridesharing and food delivery companies process streaming spatial-temporal data to timely pair supply and demand. Online advertisement platforms continuously optimize content placement to maximize monetary gains. Digital healthcare providers collect and evaluate wearable device sensory data to offer lifestyle suggestions and launch critical alarms. Data centers monitor hardware and software status around the clock to capture performance anomalies. Numerous applications seek the ability to quickly react to dynamic streaming data, as it is either a mandatory requirement or a competitive advantage.

These demands catalyze the developments of various types of stream computing engines, including Apex [2], AthenaX [10], Google Dataflow [20, 11], Flink [5, 22], Gearpump [6], Heron [18, 24], Samza [8, 31], Spark Streaming [37, 38], IBM Streams [12, 35, 28, 34, 33, 32, 26], etc. However, from the perspective of application developers, the decision is not always straightforward when exposed to abundant underlying engine options. Each stream computing engine designs their own programming API which means they are not compatible with each other.

Recently, the emerging Apache Beam [3] project aims to provide a common programming interface. The Beam SDK inherits from Google Cloud Dataflow API [20]. It is a unified model for both batch and streaming processing. Application pipelines written using the Beam API can run on different underlying engines without modifying their applications, which significantly reduces the switching overhead. Beam requires each participating engine to provide a runner that is responsible for converting Beam applications into their engine native programs and submitting the programs for execution. The Beam SDK focuses on application logic and relies on runners and engines to figure out low-level system details, such as parallelism, graph fusion, consistency, durability, and failure recovery.

This paper shares our experiences of building IBM Streams runner for Apache Beam. IBM Streams is a widely-used stream computing platform for high-throughput and low-latency applications. It has a rich set of more than 80 connectors and toolkits, many of them open-sourced [14, 12], allowing a developer to easily integrate their streaming ap-

plications with many other enterprise applications and services. It supports a broad range of programming language interfaces, including Java, C++, Python, Stream Processing Language (SPL) [27] and so on. As Beam gains increasing attention, supporting the Beam SDK further broadens the programming interfaces of IBM Streams, allowing stream applications written in Beam to take advantages of the enterprise strength of IBM Streams and its rich set of connectors and toolkits.

Streams runner traverses a Beam pipeline and creates a directed acyclic graph (DAG) that conveys the same application logic using the Java API of the open-source Streams Topology toolkit [14]. Then, the DAG can be automatically submitted to the IBM Streams Runtime or an IBM Cloud Streaming Analytics service [12]. The Streams runtime has already been optimized in the past decade in many aspects including elasticity [34], windowing efficiency [35], consistency [28], and load balancing [33]. Hence, IBM Streams runner mainly focuses on improving the efficiency of individual Beam operators.

The challenges of designing the runner lie in how to handle Beam windows and states. One challenge occurs in multi-input multi-output Beam operators whose states from input streams are interdependent. Elements from one input stream can refer to past or even future states of other input streams. When the requested states are not yet available, the operator needs to hold the element and instantly retrieve it once the state becomes ready. Efficiently indexing pending states and elements are not trivial when applications use large fan-in operators with excessively long lookahead distances. To address this problem, Streams runner maintains a specialized DAG structure, which is called Pending States Graph, to index interdependent windows. Another challenge emerges when serving an evolving set of keys. The runner has to promptly detect and delete obsolete keys to prevent hitting an out-of-memory error. To track evolving contents with minimum overhead, Streams runner dissects keyed states into smaller components, and organizes them into multiple interleaving ordered lists, which helps to delete obsolete states with low computational complexity. These two techniques collectively boost the throughput of our runner by more than 100 fold when running query 4 and 5 from Beam's NEXMark benchmark [15].

Evaluations are conducted using NEXMark, and we developed a microbenchmark to attain a more in-depth and comprehensive view of different runners. Results show that Beam applications can achieve higher throughput and lower latency when running on Streams compared to Flink and Spark.

We announced the first Streams runner GA release in October 2017 [13], which is available for download from the IBM Cloud Streaming Analytics service console [12].

This paper also summarizes lessons learned from developing the Streams runner and working with Beam applications. First, applications cannot construct a continuous count- or time-based sliding window using the existing Beam SDK. Second, mimicking continuous sliding window with dense discrete Beam sliding windows leads to miserable performance degradations. Third, the Beam programming guide guarantee that each user-defined function instance will only be executed by a single thread at a time. This means that the runner has to synchronize the entire function invocation, which could lead to significant performance bottlenecks.

The remainder of the paper is organized as follows. Section 2 briefly illustrates the background for Apache Beam and IBM Streams. Section 3 elaborates upon optimizations in Streams runner. Implementation details are described in Section 4. Section 5 shares evaluation results, and further discussions and suggestions are presented in Section 6. Section 7 summarizes related work. Finally, Section 8 concludes the paper.

2. BACKGROUND

This section briefly introduces the Apache Beam model and IBM Streams runner design.

2.1 Apache Beam

Apache Beam is a unified model for both streaming and batch data processing, which has attracted increasing attention from both academia and industry. Applications use the Beam SDK to build a pipeline (DAG) that consists of `PCollection`s and `PTransform`s. A `PCollection` can be either a bounded dataset or an unbounded data stream. A `PTransform` takes one or multiple `PCollection`s as inputs, applies user-defined functions, and produces new `PCollection`s. The Beam SDK provides a rich set of composite `PTransform`s based on six types of primitive `PTransform`s as described below.

- **`Source`** creates a new `PCollection` from a data structure or an IO connector.
- **`Window`** uses user-defined `WindowFn` to assign windows to elements.
- **`ParDo`** takes one main input `PCollection` and multiple side input `PCollection`s. It applies a user-defined `DoFn` to every element from the main input `PCollection`, and produces zero or more output `PCollection`s. The `DoFn` can access windowed states (`View`) from side input `PCollection`s.
- **`View`** converts every window of a `PCollection` into a data structure (e.g., `List`, `Map`, `Set`, `Singleton`, etc.) using an application-specified function. These data structures are later consumed as side inputs by `ParDo` `PTransform`s.
- **`GroupByKey`** groups values with the same key in the same window into a list. Applications can configure both windowing and triggering schemes.
- **`Flatten`** merges multiple input `PCollection`s of the same type into a single output `PCollection` without guarantees to preserve the arrival order of its elements.

The Beam model distinguishes event time from processing time, where event time is when the event occurs and processing time is when the element or window is processed. Every element has a timestamp meta field recording its event time. To identify late arrivals, Beam uses watermarks to estimate the progress of `PCollection`s. Every watermark is a timestamp defined in the even time domain. An element is considered late upon arrival if its timestamp falls behind the current watermark (progress) of the `PCollection`. Window boundaries are defined in the event time domain as well. Every window in Beam is subject to fixed boundaries and never changes after creation, but `PTransform`s can maintain

an evolving set of windows to support dynamic behaviors, such as sliding and tumbling paradigms. Every element belongs to one or multiple windows, determined by the element event timestamp and window boundaries. The window assignment is a meta field in the element, which travels together with the element in the pipeline. With this design, `PTransform` operators can correctly retrieve window contexts for out-of-order arrivals and update outputs accordingly.

The same Beam pipeline code can execute on various distributed computing engines, including Apache Apex [2], Google Cloud Dataflow [20], Apache Flink [5], Apache Gearpump [6], Apache Hadoop [7], Apache JStorm [1], Apache Spark [9], and IBM Streams [12, 17]. The Beam SDK specifies the computation model, and each participating engine provides a runner that converts Beam applications to engine specific programs. Beam also provides a set of integration tests to verify the correctness of runners.

2.2 IBM Streams Runner

IBM Streams [12] is a reliable distributed stream computing engine for high-throughput and low-latency applications. The Streams runner supports Apache Beam applications by translating Beam pipelines into SPL programs [27] and then executing the programs using the Streams runtime. Let us use a snippet of a sample Beam application code to illustrate the translation and execution process.

```
1  Pipeline p = Pipeline.create(options);
2  // 1. Create a source of numbers
3  p.apply("Source", GenerateSequence.from(0).withRate(
4      1, Duration.standardSeconds(1)))
5   // 2. Organize numbers into 20s FixedWindows
6   .apply("Windowing",Window.into(
7      FixedWindows.of(Duration.standardSeconds(20))))
8   // 3. Use number parity as the key
9   .apply("AssignKey", WithKeys.of((Long x) -> x % 2))
10  // 4. Group numbers based on window and key
11  .apply("GroupByKey", GroupByKey.create());
12 // Run the pipeline and get a result handle
13 PipelineResult pr = p.run();
```

This is a tiny Beam application that generates numbers and then groups them into lists based on their timestamp and parity. The first line creates an empty Beam pipeline. The following four `apply` invocations append four transforms into the pipeline. The `GenerateSequence` transform emits a series of consecutive numbers from 0, with a rate of one number per second. By default, each element uses the current system time as their timestamp when emitted from a source, and all elements are assigned to the same global window. On line 6, the `Window` transform organizes elements into tumbling windows with 20 second durations. After this transform, every element carries the boundary information of the window it belongs into. Then, the `WithKeys` transform, which is a thin wrapper of the `ParDo` transform, turns every input number `x` into a key-value pair using the user-defined function to calculate the key. Finally, the `GroupByKey` transform groups numbers with the same key (number parity in this example) in the same window into a list.

The corresponding Beam pipeline is depicted at the top in Figure 1. The Streams runner topologically traverses the Beam pipeline and creates a Java *operator* for every primitive Beam `PTransform` since all Beam-specific logic must be implemented in the Beam runner. Then, the runner relies on the Topology [14] toolkit to convert the Java program into

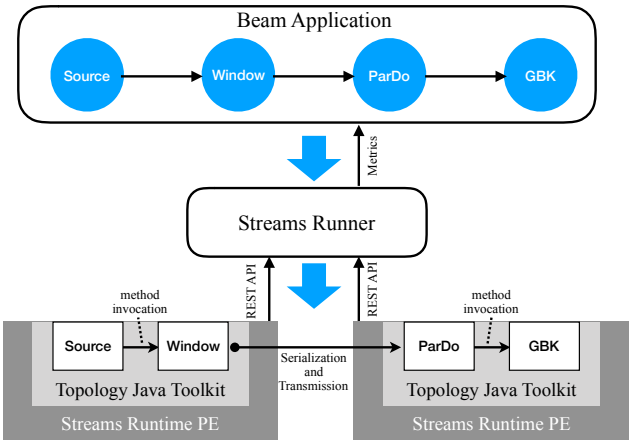


Figure 1: Architecture

an SPL program. When the program is launched, the IBM Streams runtime automatically fuses operators into processing elements (PE), which is the smallest scheduling unit in Streams. Suppose in this example, the four Java operators are fused into two PEs as shown at the bottom in Figure 1. Different PEs can run on the same or different servers, determined by the scheduling algorithm. Operators inside the same PE communicate using method invocations. Element serialization/deserialization and network transmissions only occur at PE boundaries. After submission, the application gets a `PipelineResult` object, which serves as a handle to query Beam metrics and cancel pipelines. Streams runner support Beam metrics by querying a REST API provided by the Streams runtime. Metrics are available during and after the execution of Beam pipelines.

Please refer to prior work for more details of Streams runtime designs, such as parallelism [33, 32], interface abstraction [27], windowing [35], scalability [26], elasticity [34], failure recovery [28], etc.

3. RUNNER OPTIMIZATION

This section presents three optimizations implemented in the Streams runner to improve system efficiency. Section 3.1 describes the *Pending States Graph* to index interdependent window states on multiple `ParDo` input streams. Section 3.2 describes the techniques to promptly detect and delete stale states. Section 3.3 presents how the runner applies bundles to `PCollections`.

3.1 Pending States Graph

In the Beam model, the `ParDo` transform takes one main input stream and multiple side input streams. The user-defined `DoFn` is applied to every main input element. The `DoFn` may access states from side input streams using the main input element's window information. Applications specify the mapping from a main input window to a side input window by providing a WindowMappingFn which can return past or even future windows of a side input. A side input window can flip from unready to ready randomly based on a data element arrival, or from ready to expired sequentially based on a watermark advance. The main challenge is to maintain an efficient index data structure from side input

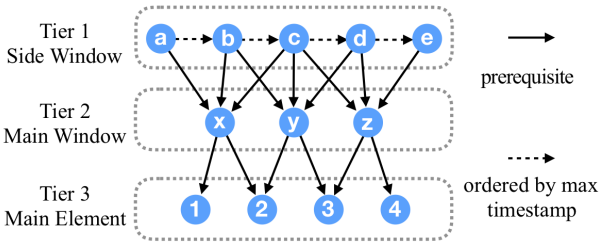


Figure 2: Pending States Graph

windows to main input element that allows both random retrieval and sequential deletion.

More specifically, Beam expects the runner to exploit all associated windows assigned to the main input element and use the `WindowMappingFn` to calculate the requested side input window for every main input window. When a specific main input element arrives, it is possible that requested side input states are not yet ready for some or all of the main input windows. In this case, the main input element together with the pending main input windows have to be temporarily pushed back and stored in the `ParDo` operator and wait for future side input data arrivals. In some applications, the amount of pushed back main input elements can be quite large since side input elements can arrive much later than the main input and nothing prevents `WindowMappingFn` from requesting a future side window. Therefore, the `ParDo` operator needs to efficiently handle side input states and pushed back main input elements.

The `ParDo` operator organizes pending states into a three-tier Pending States Graph (PSG). Figure 2 depicts an example. The first tier stores requested but unready side input windows. The second tier keeps unready main input windows. The edges pointing from the first tier to the second tier record the prerequisites to unlock main input windows. The third tier stores all pushed back elements. The edges between the second and the third tier record window assignment of pushed back elements. The `ParDo` operator inserts nodes and edges into the PSG when it encounters an unready main input window.

Later, states of a pending side input window become available when either side data arrives in that window, or side watermark passes the max timestamp of the window. To efficiently handle the latter, the first tier also internally maintains an order based on max window timestamps. When a side window becomes ready, it is deleted from the PSG together with the edges originated from it. The second and third tiers are processed in the topological order. The runner invokes `DoFn` on a main element in a window when all incoming edges of the window are removed from the PSG. Then, the window is removed from the second tier. A main element can be safely garbage collected after processing all its assigned windows. To demonstrate the algorithm, suppose the watermark advances beyond side window `b` and the operator receives data from side window `c`. Then, the PSG would drop edges from `a`, `b`, and `c`, which contain all incoming edges on main window `x`. Hence, PSG would, in turn, remove edges from `x`, which automatically activates element `1`.

Algorithm 1 shows the pseudo-code for processing a main input element. When the main input element arrives, line

Algorithm 1: Process Main Inputs

```
   Input: elem main input window
1  finished ← true
2  for w ∈ elem.windows do
3      isReady ← true
4      for s ∈ side inputs do
5          sideWindow ← s.getSideInputWindow(w)
6          if sideWindow not ready then
7              add sideWindow to tier 1 if not yet
8              isReady ← false
9      if isReady then
10         process elem in window w
11     else
12         add w to tier 2
13         finished ← false
14 if not finished then
15     push back elem to tier 3
```

2 explores its windows. For each main input window, line 4 loops over all side input windows dictated by the user-defined `WindowMappingFn`. Unready side input windows, main input windows, and main input elements are inserted into PSG accordingly on line 7, 13, 15 respectively. The two loops on line 2 and line 4 govern the computational complexity, which is $\mathcal{O}(|W|\cdot|S|)$ where $|W|$ and $|S|$ denote the number of assigned windows and the number of side inputs respectively.

Algorithm 2: Process Side Inputs

```
   Input: id side input id, w side input window
1  wid ← (id, w)
2  if wid ∈ tier 1 then
3      for mainWindow ∈ wid's neighbors in tier 2 do
4          delete the edge from wid to mainWindow
5          if mainWindow has no ingress edges then
6              for elem ∈ mainWindow.neighbors do
7                  process elem in mainWindow
8                  if elem has no ingress edges then
9                      delete elem from tier 3
10             delete mainWindow from tier 2
11     remove wid from tier 1
```

Algorithm 2 shows the pseudo-code that handles side input data. Line 1 creates a unique identifier (`wid`) for window w of side input id. On line 2, if the window exists in tier 1, then some main input is waiting for this side input window. Subsequent lines check all descendant of node `wid`, and recursively activate and remove them from PSG.

3.2 Keyed State Management

The Beam model supports stateful processing [16]. Every state (*e.g.*, user-defined states in `ParDo`, and batched elements in `GroupByKey`) exclusively binds to a specific key. In some real-world streaming applications, a key may emerge and gradually die out [30] with the passage of time. Therefore, states associated with obsolete keys have to be removed in time, otherwise the memory usage will keep increasing until the application hits an out-of-memory error. However, although keyed states are independent, their expiration may

depend on the same side input availability. Besides, as applications can create an arbitrary number of keys, the runner cannot afford to employ a dedicated heavy index data structure for every key. Hence, the challenge lies in how to timely and efficiently track key expirations. To solve this problem, we extract as many components as possible from keyed states, and manage them together, which makes the residual parts in every keyed state lightweight. To dissect keyed states, this design requires a comprehensive understanding of key expiration timings.

Three conditions collectively determine the timing for disposing obsolete states. First, a key cannot be removed if any of its windows are still active. The Beam model defines a third time concept called *watermark*. The watermark is an estimate of the progress in event time on a `PCollection`. When a watermark `W` arrives from an input stream, the operator expects all future elements from that stream have timestamps larger than `W`. Elements violating the watermark are considered late arrivals. Applications can tune lateness tolerance by setting the `allowedLateness` field in the window configuration. Late elements will not be discarded immediately. Instead, the operator only drops all elements in a window if the window falls too far behind the current watermark, which guarantees that all windows being processed have complete contexts.

Beam requires applications to provide watermarks for source `PCollection`s. Given that, in many cases, applications cannot predict future arrivals, source watermarks can be both too conservative (watermarks fall behind on-time elements) and too aggressive (introduces late arrivals). When propagating watermarks to downstream operators, runners only need to guarantee that on-time elements and their outputs never become late.

Second, as discussed in the Section 3.1, `ParDo` operators push back main input elements to wait for corresponding side inputs. Keys have to stay alive until all its pushed back elements and windows are fully processed.

Third, Beam allows user-defined `DoFn`s to use both event time timers and processing time timers. Event time timers fire when the main input stream watermark advances beyond the set time. Processing time timers, however, are triggered based on the elapsed physical time since the arrival of the first main input element in a window or set by the user. Hence, `ParDo` can only discard a key after all timers are cleared.

One straightforward design is to create a PSG for every key such that the runner can easily retrieve all pushed back elements of a key. However, this design has to loop over all keys to find pushed back elements when a side input arrives, which could become costly when the application handles many keys. To avoid this overhead, Streams runner only maintains a single PSG in each `ParDo` operator instance. The operator instance can efficiently use the PSG to locate pushed back elements and then use the keys in the elements to retrieve the keyed states. Figure 3 shows an example of the data structure. Keyed states remember the number of pushed back elements under that key. Timers are also stored in each keyed state, but are globally linked into an ordered list to avoid introducing multiple timer threads. States of a key are deleted when all associated windows violate `allowedLateness`, the corresponding counter of pushed back elements reaches zero, and all timers are cleared. Hence, the keyed states are ordered based on the smallest maximum timestamp of all windows, which allows the runner to efficient delete obsolete states on watermark arrivals.

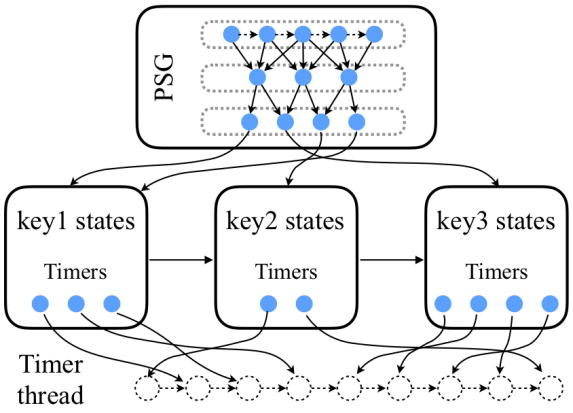


Figure 3: Keyed States

3.3 Bundling Elements

In Beam, `PCollection`s are processed in bundles. Some runners (*e.g., Flink, Spark, and Streams*) expose the interfaces for users to control the bundle size. Having smaller bundle sizes helps to reduce the batching delay but could suffer from higher amortized system overhead. However, configuring bundle sizes is different from controlling bundle boundaries, because applications can split/merge streams and discard/insert elements based on the value of input data which may not be known before execution. That is to say, applications cannot clairvoyantly predict what elements will be packed into the same bundle. Therefore, runners have the freedom to decide how to divide a `PCollection` into bundles. One solution is to enforce the bundle size at every individual operator. Each operator maintains a counter that remembers the number of processed input elements since the beginning of the bundle, and invokes the user-defined `startBundle` and `finishBundle` methods accordingly. Alternatively, the runner could enforce the bundle size only at the source. Downstream operators simply create an output bundle for all results generated after processing an input bundle.

Streams runner implements the second solution by creating a super tuple wrapping over a list of elements followed by a watermark. The first solution requires extra code to maintain bundling states in every operator, which introduces additional overhead with no added benefit. Admittedly, after a bundle leaves the source operator, downstream operators may partially process the bundle or emit fewer outputs than inputs, leading to smaller output bundle sizes. For example, user-defined `DoFn` may decide to discard main input elements, and `GroupByKey` batches input elements before the watermark reaches the max window timestamp. However, we claim that the diminished downstream bundle size does not increase the amortized system overhead. The overhead comes from two major sources: executing application `startBundle` or `finishBundle` methods, and walking through the three layers of invocations (Beam transform, Topology toolkit, and Streams Runtime).

The Streams runtime is implemented in C++. The Topology Java toolkit communicates with the Streams runtime

using JNI calls. When an element enters a PE, the Streams runtime passes the element to the Streams Topology Java toolkit by initiating a JNI up call. The JNI down calls are triggered when elements exit a PE. JNI calls, especially JNI up calls, are considerably more expensive compared to method invocations in the same language. This overhead only occurs at bundle boundaries as all elements in a bundle is transmitted as a super tuple. Therefore, for the same number of bundles, individual bundle sizes do not affect the amortized system overhead but number of bundles do. In the second solution, all operators only see a new bundle when the source operator emits one. Therefore, the amortized system overhead is well controlled by the source bundle size.

4. IMPLEMENTATION

In this section, we describe IBM Streams runner implementation details and share rationales for some implementation decisions.

4.1 Translation

IBM Streams offers the Topology toolkit [14] which allows application developers to write Streams applications using Java or Python APIs. Initially, we intended to translate Beam transforms into existing Topology operators since this would be the lightest solution. However, we soon realized that the Beam model does not align well with the Topology toolkit's APIs. Therefore, instead of using existing Topology operators, Streams runner implements new Topology operators for Beam's `Read`, `Window`, `Flatten`, `ParDo`, and `GroupByKey` transforms respectively. In this way, the runner has full control of the execution from when an element arrives at an operator until the element is emitted. Executions outside an operator (*e.g.*, thread scheduling, data transmission, etc.) are controlled by the Topology Java interfaces and Streams runtime.

4.2 ParDo Transform

`ParDo` is the most versatile and frequently used `PTransform` in Beam SDK. Therefore, its efficiency is crucial. A `ParDo`'s behavior is affected by three configurations:

- **Stateful vs stateless**: In stateful `ParDo`s, the operator needs to extract the key from every element and maintain dedicated states for that key. Besides, as described in Section 3.2, the operator also maintains an extra data structure to assist obsolete key deletions.
- **Side inputs vs no side inputs**: When side inputs are present, the operator loops over all side inputs on every element to retrieves `View` values.
- **Side outputs vs no side outputs**: When side outputs are present, the operator needs to map outputs elements to the correct output ports and broadcast watermarks to all output ports.

The above three configurations expand to eight combinations. The most efficient solution will have a dedicated implementation for every combination to avoid checking the configuration on every incoming element. However, this design will result in tedious and scattered code. To understand whether performance gains are worth the implementation complications, we compared the two designs. The first design implements six different combinations, 2 statefulness × (SISO + SIMO + MIMO) where SIMO stands for single input multiple outputs. The second design implements four combinations, consolidating SISO with SIMO. We evaluated them using Beam's official NEXMark [15] (explained in Section 5.2) benchmark under the same configuration and hardware. Figure 4 shows the results. The more concise design with four combinations takes as much as 15% longer time when processing query 1. Therefore, we decided to adopt the more complicated implementation to achieve higher efficiency.

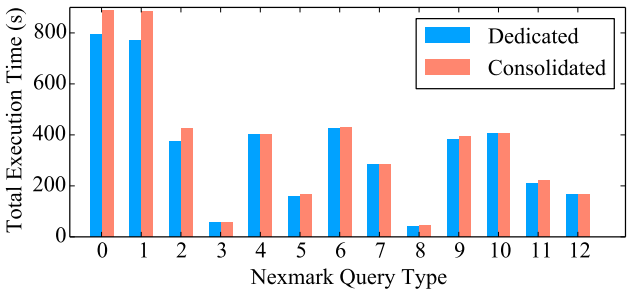


Figure 4: Consolidating ParDo Wrappers

4.3 View Transform

The `View PTransform` converts all elements in a window into a data structure by applying a user-defined data structure. Then, a `ParDo` can consume this `View` as a side input, which allows its `DoFn` to access `View` data structure when processing main input elements. There are several different options to implement this logic. The runner may route all side input elements into the `ParDo` operator and merges the `View` logic into `ParDo`, or it can create a dedicated `View` operator. Neither solution is ideal. The former introduces unnecessary overhead if the `View` is consumed by multiple `ParDo` transforms, as each `ParDo` needs to apply the grouping and user-defined `View` function to the same set of elements. The latter is more efficient, but the grouping and `View` function resemble `GroupByKey` and `ParDo` transforms. Hence, implementing a dedicated `View` operator will create code duplications. IBM Streams runner implements `View` using a third alternative. It treats the `View` primitive transform as a `Combine` composite transform which contains a `GroupByKey` and a `ParDo`. As a result, each `View` transform will be represented by two operators in the translated Streams topology.

4.4 Monitoring

IBM Streams supports monitoring with metrics that are similar but not identical to Beam metrics. Rather than create a separate metrics implementation, the IBM Streams runner implements Beam metrics on top of IBM Streams custom metrics. This allows existing IBM Streams tooling to monitor Beam application metrics, but does cause some limitations and complications for the implementation due to the slightly different semantics.

IBM Streams custom metrics can be created and manipulated by IBM Streams applications. Custom metrics types include counters, times, and gauges, but gauges do not have timestamps as they do in Beam. While the IBM Streams

runtime uses a metric type called histogram that could represent Beam distributions this metric type is not available to applications as custom metrics.

To support Beam gauges and distributions metrics, multiple IBM Streams metrics are used to represent a single Beam metric. Beam gauges are represented with two IBM Streams custom metrics, a gauge and a time. The implementation of Beam distributions uses four IBM Streams custom metrics: two counters for the count and sum, and two gauges for the min and max.

IBM Streams does not provide for atomic updates of multiple metrics, which means that there can be instances where the Beam metric briefly appears inconsistent. While the update is done in order to try to minimize this effect, metrics consumers should be aware of the possibility.

Much like Beam metrics are per-step, IBM Streams custom metrics are per-operator. As described above, Beam pipelines are transformed into custom Topology operators for IBM Streams, but these operators do not always correspond one-to-one with Beam steps. In some instances, such as transform fusion, a single operator can implement multiple Beam steps, so the runner needs to be able to separate the metrics by step. In the case of pipeline parallelism, a single transform may be running in multiple parallel operator instances with separate metrics.

To handle these cases and to keep Beam metrics visually distinct from other IBM Streams metrics, the IBM Streams runner encodes Beam metrics in a human-readable form that includes the Beam step, namespace, and metric name, as well as the type of metric and what element of multipart metrics it represents. This allows the runner to both aggregate metrics from parallel instances, and separate metrics by their Beam step. It also allows newer IBM Streams tooling such as its web console to be Beam-aware and render Beam metrics appropriately while still allowing users to understand the metrics and their relation to Beam even in their raw form.

The IBM Streams runner currently only implements attempted metrics, it does not implement committed metrics.

A final difference between Beam metrics and IBM Streams metrics is when they are available. Beam metrics are an optional feature so may not be available at all but even when they are they may not be available during runtime; this is a runner-dependent choice. They are usually available when the pipeline finishes. IBM Streams pipelines normally run continuously until cancelled or the application fails, so its metrics are only available during runtime and they are not preserved at pipeline termination.

The IBM Streams runner makes Beam metrics available during runtime by using the IBM Streams REST API to retrieve metrics. It will also poll all metrics when the application cancels itself, but if the application is cancelled externally or fails, metrics will not be available.

Overall, this approach to implementing Beam metrics with IBM Streams metrics has been successful. While there are minor limitations, the ability to reuse existing tooling has proven useful.

4.5 Watermark

Beam employs watermark as the estimate of progress in event time domain. The watermark triggers timers in `ParDos` and windows in `GroupByKeys`. Unbounded `Sources` in Beam provide an `getWatermark` API. Applications can customize `Sources` and implement `getWatermark` accordingly. The source operator in Streams runner polls the `getWatermark` API after reading any element from the application `Source` and emits the watermark if it is larger than the last value. Beam does not specify how runners should propagate watermarks in non-`Source` operators. Streams Beam runner follows the design presented in Dataflow [20] where the output watermark of an operator is the minimum watermark of all input streams.

Table 1: Evaluation Hardware Specifications

OS	Red Hat Enterprise Linux Server 6.9
Kernel	2.6.32-696.3.1.el6.x86_64
CPU	Architecture: x86_64 CPU(s): 16 Thread(s) per core: 2 Core(s) per socket: 4 Socket(s): 2 NUMA node(s): 2 CPU family: 6 Model: 44 Model name: Intel(R) Xeon(R) CPU X5687 CPU MHz: 3599.735 NUMA node0 CPU(s): 0-3,8-11 NUMA node1 CPU(s): 4-7,12-15
Memory	available: 2 nodes (0-1) node 0 size: 32756 MB node 1 size: 32768 MB node distances: node 0 1 0: 10 21 1: 21 10
Network	Broadcom BCM5709S Gigabit Ethernet

5. EVALUATION

In order to compare various stream computing systems, prior works have to implement the same application using completely disparate APIs [38]. However, evaluation results may not be convincing as the implementation efficiency often relates to the proficiency on the target system. One can rarely, if ever, argue that different implementations of the same application are identically optimized. The emergence of Apache Beam eliminates this concern because the same application code is able to be run on different engines. The diverging system performance completely attributes to the runner and the underlying engine which are both considered components of the holistic stream computing solution.

This section compares Streams against Flink and Spark. Flink and Spark runners were selected because they have been available since Beam's incubation releases, bundle size and bundle time interval can be configured, and they can be deployed within our uniform performance environment. Another mature runner is Google Dataflow; however, this runner is designed to submit Beam pipelines to the Google Cloud Platform which we cannot setup in our environment.

5.1 Environment

Evaluations were conducted across four identical systems using Beam 2.2 packages, an IBM Streams 4.2.4.0 instance, local Flink 1.3.0, and local Spark 1.6.3 runtimes. Besides

bundle size and interval parameters, default runner options were used. Table 5 describes the server specifications used.

5.2 NEXMark

Since the 2.2.0 release, Beam SDK natively provides an implementation for the NEXMark benchmark [15]. NEXMark mimics an online auction application, and defines thirteen types of queries over three data streams, namely Person stream, Auction stream, and Bid stream. The queries are originally defined using the Continuous Query Language (CQL) [21]. Please refer to [15] for their CQL code and detailed explanations. In general, the first three queries use simple stateless transforms or filters, while the remaining ten queries involve more complex operators, such as join and aggregation.

Beam SDK implements these 13 NEXMark queries inside of a framework which surfaces several configuration parameters. Since we are evaluating streaming engines, all runners were configured for streaming mode, and the framework was configured to collect performance metrics while the job was running. Pipelines were given 30 seconds before performance metrics were collected to minimize initialization bias.

Beam's NEXMark framework, by default, summarizes 10 samples or two minutes data to compute steady-state performance metric measurements. To closely achieve steady-state event rates in a timely manner, individual NEXMark queries were launched to consume a large number of events. The detailed numbers of events are shown in Table 2.

Table 2: NEXMark Query Source Configuration

Query	Number of Events
0-2	25,000,000
3-11	2,000,000
12	500,000

Figure 5 shows the results. All experiments configure bundle batching time to 100 ms. The three figures show the throughputs when setting the bundle size to 1, 10, and 100 elements respectively. The throughput clearly increases for all runners when using larger bundle sizes. Compared to Flink, Streams attains much higher throughput in all queries except query 10 and 12. Outperforming in query 0 means that the Streams achieves much lower system overhead compared to Flink. The mixed result in the other queries indicates that Streams runner is not perfectly optimized yet, and it limps at certain types of queries. This result aligns with our expectation. In query 12 where Streams runner suffers, the pipelines employ `Combine` transforms or repeatedly use `GroupByKey` followed by `ParDo` patterns. Flink runner has already implemented a `Combine` optimizer which conducts the grouping and combining functions in the same operator. In contrast, without a dedicated `Combine` optimizer, Streams runner still treats `Combine` as a normal composite transform, and translates it into a `GroupByKey` transform and a `ParDo` transform. Spark Streaming staggers miserably under the configured batch sizes. As its results are almost invisible from the figures, we use black arrows to point out the performance numbers for query 2 and 9. Spark streaming generates an RDD for every timestep bundle, and submits it as a batch job to the Spark core. For tiny bundle sizes, this design endures huge system overhead, and hence leads to extremely low throughputs. Besides, Spark runner cannot execute query 3 and 7 yet, as it does not support states, timers, or the `View` `PTransform` in streaming mode. For more details regarding these missing features, please refer to Beam-1035, Beam-1115, and Beam-2112 on Beam's Jira site.

One limitation of Beam's NEXMark framework is the lack of detailed per-tuple response time metrics. Additionally, the Flink runner does not support Beam metrics while the pipeline is running. Instead, the runner only displays statistics after the pipeline finishes. Therefore, we developed a couple micro benchmarks to collect more comprehensive performance measurements.

5.3 Micro Benchmark

Two micro benchmarks were created to measure latencies and throughputs under various bundle sizes and bundle time intervals. These benchmarks focus on the performance of the `ParDo` and `GroupByKey` transforms, as they are the most versatile and expensive components in Beam pipelines. Per tuple latencies and average throughputs were collected against the IBM Streams and Apache Flink runners. Each benchmark, runner, configuration test was conducted 20 times. This set of evaluations skip Spark because the previous results in Section 5.2 have already shown that Spark Streaming cannot deliver comparable results under small bundle configurations.

5.3.1 Benchmark Pipeline

Each benchmark pipeline starts from an `UnboundedSource` transform generating an endless amount of elements (24 bytes in size) as quickly as possible.

To measure the tuple latency and overall throughput, each element is given a timestamp before it is provided to the source's reader. A final reporting `PTransform` is appended at the end of the micro benchmark pipeline to process the element's timestamp and record the latency. To remove any initialization bias, the reporter starts recording latencies after 30 seconds into the experiment.

5.3.2 ParDo Benchmark

The `ParDo` micro benchmark pipeline consists of 20 sequential `ParDo` transforms that pass elements directly from input to output conducting no other work. Results from this micro benchmark indicate element transportation time/cost and show how efficiently runners transport elements between transforms.

Figure 6 (a) shows the evaluation results when the bundle time interval is configured to an excessively large value (1000 seconds) such that the bundle size limit is always triggered first. The per-tuple latency increases with the increase of bundle size, which meets with our expectation as larger bundle sizes mean longer batching delay in Streams runner. However, the same benchmark shows a completely reversed trend on Flink where the per-tuple latency decreases with the increase of bundle size. To understand this counter-intuitive phenomenon, we dug into Flink runner's source code, and found that it does not treat a bundle as a batch. Flink emits a tuple immediately instead of waiting for an entire bundle to finish, unless a snapshot is under construction. The bundle size limits are only used for triggering user-defined `startBundle` and `finishBundle` methods. Therefore, increased bundle size does not contribute to per-tuple latency, but the overheads of `startBundle` and `finishBundle`

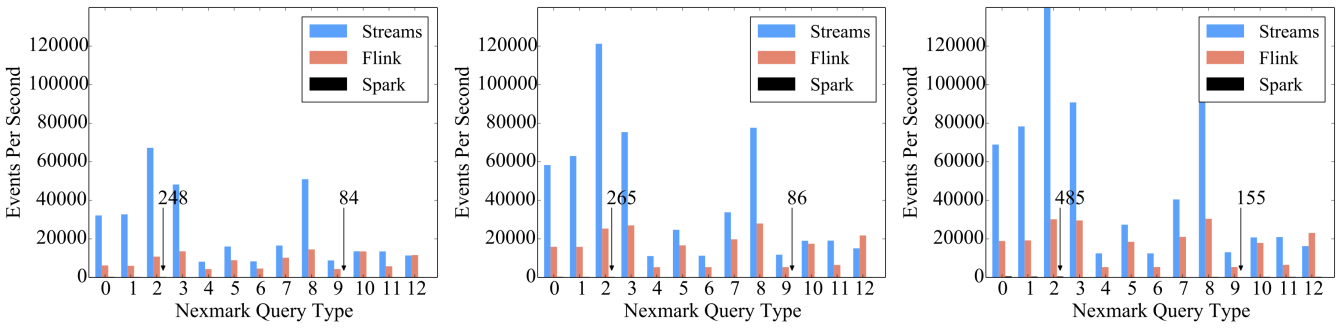


Figure 5: Nexmark Benchmark Evaluation

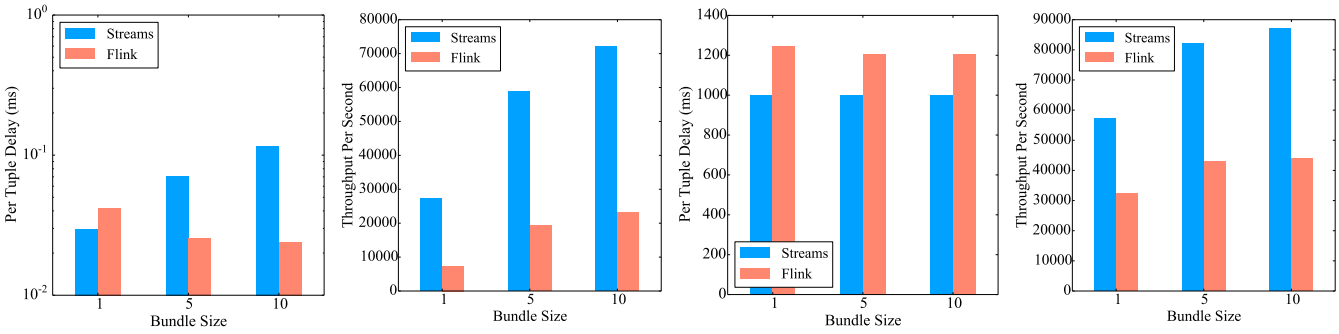


Figure 6: Micro Benchmark: Fixed large bundle time millis, varying bundle sizes

are amortized across more tuples, leading to a lower per-tuple latency. Although bundles do not have to align with batches, this design in Flink runner lacks the option for applications to trade batch sizes for system throughputs when micro-second latency is overkill. Figure 6 (a) also shows that Streams achieves 4 times higher throughput than Flink.

Then, in Figure 7 (a), the bundle size is set to a excessively large value (10^7) while various bundle time intervals are compared. Configured bundle time intervals are highlighted by the horizontal dashed lines in the figure. Again, as Flink does not treat the bundle time interval as the batch time interval, it achieves much lower per-tuple latency. The latency in Streams is about 20 times the configured bundle time interval, which is expected as the Beam pipeline contains 20 sequential `ParDo` transforms. The throughput measurements also show Streams outperforms Flink.

5.3.3 *GroupByKey Benchmark*

The `GroupByKey` micro benchmark consists of several sequential transforms including applying windows to generated elements, assigning keys to the elements, grouping elements by key, and splitting grouped elements back to individual items. For this instance, sliding windows of two second duration and one second period were applied to the elements and keys were assigned by round-robin across 10 unique keys.

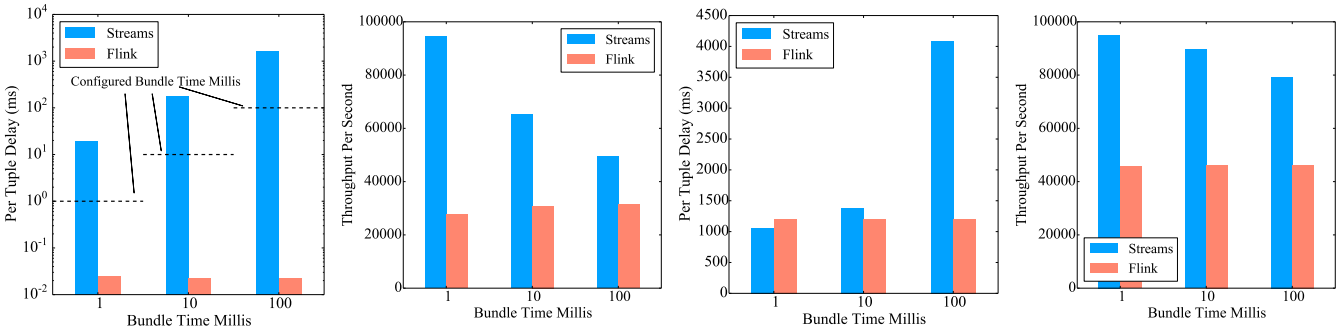
The first set of evaluations fix the bundle time interval to 1000 second and test across various bundle sizes. Results are presented in Figure 6 (b). The per-tuple latency in streams is close to the application windowing delay of 1000 ms. On average, Flink spends about 200 ms longer to emit a keyed window output compared to Streams. Streams achieves about 2 times throughput compared to Flink. The second set of evaluations fix the bundle size to a large value (10^7) and vary the bundle time interval. As Flink does not batch tuples based on bundle size, its per-tuple latency stays roughly the same value. The latency in Streams slightly increases when using 10 ms bundle time interval, but significantly jumps 4 times when using 100 ms due to larger bundle time intervals causing more elements per bundle. As the runner only appends a watermark at the end of every bundle, states within a bundle cannot be timely emitted and deleted, leading to a much higher state management overhead. If read together with the `GroupByKey` throughput result in Figure 6 (b), we can calculate that each bundle contains roughly 8,000 elements when bundle time interval is set to 100 ms. Although Streams still manages to outperform Flink by 2 times in throughput, the gap shrinks as the time interval increases. We believe this can be remedied by inserting more watermarks into large bundles.

6. LESSONS LEARNED

During the design and development of Streams runner, we gradually comprehended the strengths and weaknesses of the Beam model. In this section, we share several application scenarios that Beam fails to accomplish efficiently. When encountering these requirements, the runner and the application developers have to consider competent solutions.

6.1 Continuous Window Model

Windowing is a fundamental technique in stream processing engines. Existing window models can be categorized



(a) ParDo (b) GroupByKey

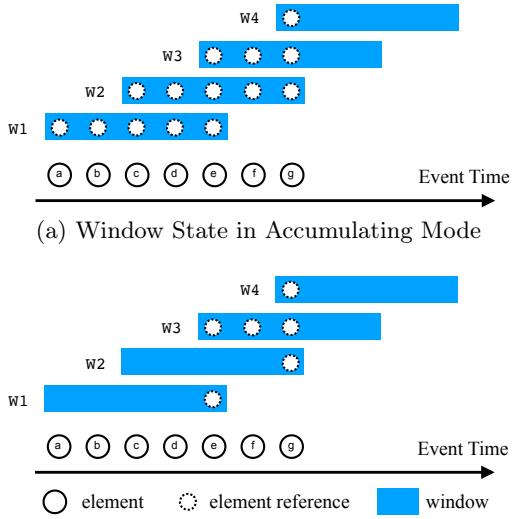
Figure 7: Micro Benchmark: Fixed large bundle size, varying bundle time intervals

into two types: continuous and discrete. Continuous window models maintain a single window that slides along the time axis, while discrete window models keep track of an evolving set of static windows where every window covers a fixed time interval. The entire set of windows travels along the time axis by creating new windows and discarding obsolete windows. A continuous window model maintains fewer states than a discrete window model, which leads to higher efficiency but on the other hand lacks the support for late arrivals. Even if it keeps past window contexts, a late element could intersect with a formidable number of past windows, leading to an unacceptable overhead. In contrast, discrete windows usually align outputs at sporadic window boundaries, resulting in a more affordable set of window contexts for late arrivals. Apache Beam adopts the discrete window model.

However, discrete window models also find themselves clumsy when serving applications that detect certain patterns from a continuing series of signals. For example, consider a wearable device using an EKG [25] sensor to monitor cardiac readings, or a trading algorithm that keeps track of market data within the past one minute. Beam applications will have to split the signal sequence into discrete windows, which might tear the target pattern apart and result in false negatives. Moreover, as window outputs are only generated at sporadic time instances, the Beam model falls short for applications seeking extremely low response times. Regardless of how dense an application configures Beam's sliding window, it never becomes fully continuous. Using dense windows leads to another concern as described in the following section.

6.2 Dense Sliding Windows

Although the Beam API covers a broad spectrum of scenarios, application developers need to take care if they generate frequent (*e.g.*, 1 second) outputs from relatively large (*e.g.*, 5 minute) windows due to inefficiencies in Beam's window state management algorithm. In Beam's design, every window is associated with a dedicated data structure that holds all elements in the window. As windows with large sizes and small periods excessively overlap with each other, a single input element may need to be added into a large number of windows resulting in redundant states and computations. Figure 8 (a) shows an example where window size equals five and window period two. One element arrives in every time unit, which will be inserted into two or three windows.



(a) Window State in Accumulating Mode

(b) Discarding Mode Triggered on the 4^{th} Element

Figure 8: Beam Window State

Moreover, some of Beam features prevent runners from fully optimizing these use cases. For instance, the Beam API allows applications to configure refinement policies in windows, which can be either accumulating or discarding in Beam. The refinement policy controls how the engine should handle past states after firing a window. In the accumulating mode, elements arrived after firing window will be processed together with all existing elements in the window. In contrast, the discarding mode forces the engine to clear states in a window once they are processed, and subsequent arrivals in the window start from an empty state. As different windows fire at different times, overlapping windows have to maintain their overlapping parts separately. Figure 8 (b) shows a discarding mode example of windows triggered on every 4^{th} element. In this example, `W1`, `W2`, and `W3` overlap with each other, but the overlapping event time regions are in different states.

6.3 Parallel Processing

Stateless transforms (*e.g.*, `Window`, `Flatten`, and stateless `ParDo`) can easily parallelize to multiple worker instances. Beam also supports stateful processing [16]. It requires the stateful `ParDo` and `GroupByKey` to consume a key-value main

input stream. Rather than maintaining a global state for all elements, every key has its dedicated states in Beam. With this design, a runner can safely partition a stream based on the key and execute the `ParDo` in a distributed fashion. However, Beam does not guarantee thread-safety for multiple threads to concurrently access the same transform instance regardless of whether they maintain states. Instead, Beam promises applications that there will only be a single thread executing their user-defined functions [4] at a time. Therefore, if the underline engine spawns multiple threads, the runner has to synchronize the entire `DoFn` or `GroupByKey` invocation.

Before discussing the impact of synchronizations in Beam transforms, let us first explain the threading model in IBM Streams. When executing a Streams application, the source operator asks the runtime to create the number of threads it needs. Instead of binding to a single operator, threads travel across multiple operators following the links on the pipeline, and returns to the source operator to process the next element after finishing the prior one. Hence, there are two parallelism concepts in a PE, pipeline parallelism, and operator parallelism. The former refers to having multiple threads executing on the same pipeline, while the latter runs multiple threads on the same operator instance.

As Beam forbids multiple threads from entering the same `PTransform` instance, engines lose the opportunity to use operator parallelism. It is not terribly inefficient as long as computational overhead is well balanced among all operators. However, in IBM Streams use cases, there are indeed very heavy custom operators, which correspond to expensive `DoFns` in Beam. These operators run on powerful servers with hundreds of cores. If translated to Beam, those costly `DoFns` cannot scale up to harvest the power of parallelism for (computationally) heavy and (quantitatively) hot keys. Expensive `DoFns` also impede pipeline parallelism, as threads have to compete to get through them.

Admittedly, in some scenarios, synchronizing the entire `DoFn` is absolutely necessary. But in many other cases, custom operators or user-defined `DoFns` consist of thread-safe pre-processing and post-processing steps wrapping around the real critical section. Hence, splitting the `DoFn` into smaller pieces could further help runners and engines to harvest the power of parallelism.

7. RELATED WORK

Apache Beam was hatched from the programming model/SDK portion of Google Cloud Dataflow [11]. It was designed to handle unbounded, unordered, and global-scale datasets. Akidau *et al.* discussed model details and the implementation of then underlying MillWheel stream processing system in literature [20, 19].

Several stream computing engines have joined the Beam runner community so far. Apache Flink [5] is one of the earliest participants. Carbone [22] explains the consistent distributed snapshot algorithm employed in Fink state management module, which is similar to the classical Chandy-Lamport's protocol [23]. Flink's distributed snapshot implementation address circles in the computation graph by assigning special `IterationHead` and `IterationTail` into every circle. LinkedIn developed Samza [8] and uses it for many production applications. Noghabi *et al.* [31] introduces Samza's high-level design. Compared to Flink, instead of using distributed snapshots, Samza recovers from failure by replaying change logs. It speeds up recoveries by employing a concept called *Host Affinity*, which schedules the restarted operator to where the change logs are locally stored in a best effort manner. However, without a distributed snapshot, Samza's failure recovery solution cannot guarantee global consistency. Twitter started processing stream data with Storm [36], and then switched from Storm to Heron [18]. Storm [36] allows applications to construct a computation topology with *spouts* (sources) and *bolts* (internal nodes). It can then automatically partition the application and run it in a distributed manner. Later, Storm was upgraded to Heron [18] with enhanced scalability and performance. Compared to Storm, supporting back pressure is one of the major improvements in Heron [29]. With back pressure, downstream bolts can influence the rate at which upstream bolts/spouts emit data and therefore avoid dropping data due to the large backlog. Floratou *et al.* [24] created a variant from Heron, which focuses on self-tuning, self-stabilizing, and self-healing.

All systems described above are able to promptly deliver updates on every incoming tuple. Spark Streaming behaves differently as it carries out computations based on micro-batches. Spark Streaming sits on top of Spark core. The latter is an in-memory distributed batch computing system. Datasets in Spark [37] are organized into immutable resilient distributed datasets (RDD). Applications create lineage graphs where every node in the graph is an RDD and every edge a transformation. The Spark engine divides the lineage graph into smaller components by using shuffling transformations as boundaries. Every component will be organized into a stage which is further divided into parallel tasks. Intermediate results within a stage are kept in memory, but data generated at stage boundaries need to be committed onto disks. Spark Streaming [38] divides data stream into time steps, and creates an RDD for every time step. Then it submits a Spark batch job for every time step RDD. This layer of abstraction equips Spark with the ability to handle data streams. However, by using the underlying batch computing engine, Spark Streaming can hardly deliver per-tuple updates. Forcing per-tuple time step will result in miserable throughput.

IBM Streams recently releases its runner to support Beam. During the past decade, IBM Streams has been optimized in many aspects. Tangwongsan *et al.* [35] designed the *Reactive Aggregation*, which is a framework to help Streams handle incremental sliding-window aggregations highly efficiently. Schneider and Wu [34] introduced a scalable and elastic operator scheduler to automatically figure out the number of threads to use for arbitrary sized applications. Schneider *et al.* [32] designed an algorithm that can judiciously identify data parallel opportunities in a general stream processing graph. Gedik *et al.* [26] answered the question of how many parallel channels provide the best throughput. IBM Streams [33] balances load for data-parallel regions by using TCP blocking rate per connection as the feedback signal. More specifically, it creates a function for each connection based on this blocking rate, and minimizes the maximum value of that function across all parallel channels. Guaranteed tuple processing [28] is achieved by periodically running a variation of the Chandy-Lamport snapshot algorithm [23] to capture a consistent view of global states. Thanks to prior efforts committed to IBM Streams, the development

of Streams runner only needs to focus on efficiency within individual Java operators.

8. CONCLUSION AND FUTURE WORK

This paper discussed the design, implementation, and evaluation of IBM Streams runner for Apache Beam. Streams runner converts Beam applications into Streams SPL programs by calling the open-source Topology Java toolkit, creating one Java operator for every Beam `PTransform`. We spend most of the development efforts on optimizing the efficiency in every operator. Evaluations show that the same Beam NEXMark benchmark finishes with lower latency and higher throughput on Streams compared to Flink and Spark. As an open and rapidly iterating project, Apache Beam unifies many distributed computing systems. IBM Streams offers its runner for developers to utilize the Beam SDK alongside the Streams Topology Java and Python APIs and wishes to see the flourishing programming abstractions enable and cultivate the shift to event driven architectures and real-time analytics. In the near future, we plan to focus on optimizing parallelism and failure recovery in the runner, as well as adopting new features from Apache Beam.

9. REFERENCES

[1] Alibaba JStorm: an enterprise fast and stable streaming process engine. http://jstorm.io/. Retrieved Feb, 2018.

[2] Apache Apex: Enterprise-grade unified stream and batch processing engine. https://apex.apache.org/. Retrieved Feb, 2018.

[3] Apache Beam: An advanced unified programming model. https://beam.apache.org. Retrieved Feb, 2018.

[4] Apache Beam Programming Guide. https://beam.apache.org/documentation/programming-guide/. Retrieved Feb, 2018.

[5] Apache Flink: an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications. https://flink.apache.org/. Retrieved Feb, 2018.

[6] Apache Gearpump: a real-time big data streaming engine. https://gearpump.apache.org. Retrieved Feb, 2018.

[7] Apache Hadoop. http://hadoop.apache.org/. Retrieved Feb, 2018.

[8] Apache Samza: a distributed stream processing framework. https://samza.apache.org/. Retrieved Feb, 2018.

[9] Apache Spark: a fast and general engine for large-scale data processing. https://spark.apache.org/. Retrieved Feb, 2018.

[10] AthenaX: SQL-based streaming analytics platform at scale. http://athenax.readthedocs.io/. Retrieved Feb, 2018.

[11] Google Dataflow: Simplified stream and batch data processing, with equal reliability and expressiveness. https://cloud.google.com/dataflow/. Retrieved Feb, 2018.

[12] IBM Stream Analytics: Leverage continuously available data from all sources to discover opportunities faster. https://www.ibm.com/cloud/streaming-analytics. Retrieved Feb, 2018.

[13] IBM Streams Runner for Apache Beam. http://ibmstreams.github.io/streamsx.documentation/docs/beamrunner/. Retrieved Feb, 2018.

[14] IBM Streams Topology Toolkit. http://ibmstreams.github.io/streamsx.topology/. Retrieved Feb, 2018.

[15] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/nexmark/. Retrieved May, 2018.

[16] Stateful processing with Apache Beam. https://beam.apache.org/blog/2017/02/13/stateful-processing.html. Retrieved Feb, 2018.

[17] StreamsDev: IBM Streams Developer Community. https://developer.ibm.com/streamsdev/. Retrieved Feb, 2018.

[18] Twitter Heron: A realtime, distributed, fault-tolerant stream processing engine from Twitter. https://twitter.github.io/heron/. Retrieved Feb, 2018.

[19] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[20] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[21] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *PVLDB*, 15(2):121–142, 2006.

[22] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink: consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[23] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

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

[25] T. R. Fulford-Jones, G.-Y. Wei, and M. Welsh. A portable, low-power, wireless two-lead ekg system. In *Engineering in Medicine and Biology Society, 2004. IEMBS'04. 26th Annual International Conference of the IEEE*, volume 1, pages 2141–2144, 2004.

[26] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[27] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, et al. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[28] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. Consistent regions: Guaranteed tuple processing in ibm streams. *PVLDB*, 9(13):1341–1352, 2016.

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

[30] S. Li, S. Hu, R. Ganti, M. Srivatsa, and T. Abdelzaher. Pyro: A spatial-temporal big-data storage system. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 97–109, 2015.

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

[32] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Safe data parallelism for general streaming. *IEEE transactions on computers*, 64(2):504–517, 2015.

[33] S. Schneider, J. Wolf, K. Hildrum, R. Khandekar, and K.-L. Wu. Dynamic load balancing for ordered data-parallel regions in distributed streaming systems. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, 2016.

[34] S. Schneider and K.-L. Wu. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 648–661, 2017.

[35] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[38] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.