StreamBox: Modern Stream Processing on a Multicore Machine

Hongyu Miao[1], Heejin Park[1], Myeongjae Jeon[2],
Gennady Pekhimenko[2], Kathryn S. McKinley[3], and Felix Xiaozhu Lin[1]
[1]*Purdue ECE* [2]*Microsoft Research* [3]*Google*

Abstract

Stream analytics on real-time events has an insatiable demand for throughput and latency. Its performance on a single machine is central to meeting this demand, even in a distributed system. This paper presents a novel stream processing engine called StreamBox that exploits the parallelism and memory hierarchy of modern multicore hardware. StreamBox executes a pipeline of transforms over records that may arrive out-of-order. As records arrive, it groups the records into ordered epochs delineated by watermarks. A watermark guarantees no subsequent record's event timestamp will precede it.

Our contribution is to produce and manage abundant parallelism by generalizing out-of-order record processing within each epoch to out-of-order epoch processing and by dynamically prioritizing epochs to optimize latency. We introduce a data structure called *cascading containers*, which *dynamically* manages concurrency and dependences among epochs in the transform pipeline. StreamBox creates sequential memory layout of records in epochs and steers them to optimize NUMA locality. On a 56-core machine, StreamBox processes records up to 38 GB/sec (38M Records/sec) with 50 ms latency.

1 Introduction

Stream processing is a central paradigm of modern data analytics. Stream engines process unbounded numbers of records by pushing them through a pipeline of *transforms*, a continuous computation on records [3]. Records have *event* timestamps, but they may arrive out-of-order, because records may travel over diverse network paths and computations on records may execute at different rates. To communicate stream progression, transforms emit timestamps called *watermarks*. Upon receiving a watermark w_{ts}, a transform is guaranteed to have observed all prior records with *event* time $\leq ts$.

Most stream processing engines are distributed because they assume processing requirements outstrip the capabilities of a single machine [38, 28, 32]. However, modern hardware advances make a single multicore machine an attractive streaming platform. These advances include (i) high throughput I/O that significantly improves ingress rate, e.g., Remote Direct Memory Access (RDMA) and 10Gb Ethernet; (ii) terabyte DRAMs that hold massive in-memory stream processing state; and (iii) a large number of cores. This paper seeks to maximize streaming throughput and minimize latency on modern multicore hardware, thus reducing the number of required machines to process streaming workloads.

Stream processing on a multicore machine raises three major challenges. First, the streaming engine must extract parallelism aggressively. Given a set of transforms $\{d_1, d_2, \cdots, d_n\}$ in a pipeline, the streaming engine should exploit (i) pipeline parallelism by simultaneously processing all the transforms on different records in the data stream and (ii) data parallelism on all the available records in a transform. Second, the engine must minimize thread synchronization while respecting dependences. Third, the engine should exploit the memory hierarchy by creating sequential layout and minimizing data copying as records flow through various transforms in the pipeline.

To address these challenges, we present StreamBox, an out-of-order stream processing engine for multicore machines. StreamBox organizes out-of-order records into *epochs* determined by *arrival time* at pipeline ingress and delimited by periodic *event time* watermarks. It manages all epochs with a novel parallel data structure called *cascading containers*. Each container manages an epoch, including its records and end watermark. StreamBox dynamically creates and manages multiple inflight containers for each transform. StreamBox links upstream containers to their downstream consuming containers. StreamBox provides three core mechanisms:

(1) StreamBox satisfies dependences and transform

correctness by tracking producer/consumer epochs, records, and watermarks. It optimizes throughput and latency by creating abundant parallelism. It populates and processes multiple transforms and multiple in progress containers per transform. For instance, when watermark processing is a long latency event, StreamBox is not stalled, because as soon as any subsequent records arrive, it opens new containers and starts processing them.

(2) StreamBox elastically maps software parallelism to hardware. It binds a set of worker threads to cores. (i) Each thread independently retrieves a set of records (a *bundle*) from a container and performs the transform, producing new records that it deposits to a downstream container(s). (ii) To optimize latency, it prioritizes the processing of containers with timestamps required for the next stream output. As is standard in stream processing, outputs are scoped by *temporal windows* that are scoped by watermarks to one or more epochs.

(3) StreamBox judiciously places records in memory by mapping streaming access patterns to the memory architecture. To promote sequential memory access, it organizes pipeline state based on the output window size, placing records in the same windows contiguously. To maximize NUMA locality, it explicitly steers streams to flow within local NUMA nodes rather than across nodes.

We evaluate StreamBox on six benchmarks with a 12-core and a 56-core machines. StreamBox scales well up to 56 cores, and achieves high throughput (millions of records per second) and low latency (tens of milliseconds) on out-of-order records. On the 56-core system, StreamBox reduces latency by a factor of 20 over Spark Streaming [38] and matches the throughput of results of Spark and Apache Beam [3] on medium-size clusters of 100 to 200 CPU cores for *grep* and *wordcount*.

The full source code of StreamBox is available at `http://xsel.rocks/p/streambox`.

2 Stream model and background

This section describes our out-of-order stream processing model and terminology, summarized in Table 1.

Streaming pipelines A stream processing engine receives one or more streams of *records* and performs a sequence of transforms $\mathscr{D} = \{d_1, d_2, \cdots, d_n\}$ on the records $\mathscr{R}$. Each record $r_{ts} \in \mathscr{R}$ has a timestamp ts for temporal processing. A record has an *event* timestamp defined by its occurrence (e.g., when a sensor samples a geolocation). Ingress of a record to the stream engine determines its *arrival* timestamp.

Out-of-order streaming Because data sources are diverse, records travel different paths, and transforms operate at different rates, records may arrive *out-of-order* at the stream processing engine or to individual transforms.

Table 1: Terminology

Term	Definition
Stream	An unbounded sequence of records
Transform	A computation that consumes and produces streams
Pipeline	A dataflow graph of transforms
Watermark	A special event timestamp for marking stream progression
Epoch	A set of records arriving between two watermarks
Bundle	A set of records in an epoch (processing unit of work)
Evaluator	A worker thread that processes bundles and watermarks
Container	Data structure that tracks watermarks, epochs, and bundles
Window	A temporal processing scope of records

To achieve low latency, the stream engine must *continuously* process records and thus cannot stall waiting for event and arrival time to align. We adopt the out-of-order processing (OOP) [27] paradigm based on *windows* to address this challenge.

Watermarks and stream epochs Ingress and transforms emit strictly monotonic event timestamps called *watermarks* w_{ts}, as exemplified in Figure 1(a). A watermark guarantees no subsequent records will have an event time earlier than ts. At ingress, watermarks delimit ordered consecutive *epochs* of records. An epoch may have records with event timestamps greater than the epoch's end watermark due to out-of-order arrival. The stream processing engine may process records one at a time or in *bundles*.

We rely on stream sources and transforms to create watermarks based on their knowledge of the stream data [2, 3]. We do not inject watermarks (as does prior work [7]) to force output and manage buffering.

Pipeline egress Transforms define event-time *windows* that dictate the granularity at which to output results. Because we rely on watermarks to define streaming progression, the *rate* of egress is bounded by the rate of watermarks, since a transform can only close a window after it receives a watermark. We define the *output delay* in a pipeline from the time it first receives the watermark w_{ts} that signals the completion of the current window to the moment when it delivers the window results to the user. This critical path is implicit in the watermark timestamps. It includes processing any remaining records in epochs that precede w_{ts} and processing w_{ts} itself.

Programming model We use the popular model from timely dataflow [30], Google dataflow [3], and others. To compose a pipeline, developers declare transforms and define dataflows among transforms. This is exemplified by the following code that defines a pipeline for Windowed Grep, one benchmark used in our evaluation (§9).

```
// 1. Declare transforms
Source<string> source(/*config info*/);
FixedWindowInto<string> fwi(seconds(1));
WindowedGrep<string>wingrep(/*regexp*/);
Sink<string> sink();

// 2. Create a pipeline
Pipeline* p = Pipeline::create();
```

```
p->apply(source); //set source

// 3. Connect transforms together
connect_transform(source, fwi);
connect_transform(fwi, wingrep);
connect_transform(wingrep, sink);

// 4. Evaluate the pipeline
Evaluator eval(/*config info*/);
eval.run(p);  // run the pipeline
```

Listing 1: Pseudo code for Windowed Grep pipeline

To implement a transform, developers must define the following functions, as shown in Figure 1(b): (i) `ProcessRecord(r)` consumes a record `r` and may emit derived records. (ii) `ProcessWm(w)` consumes a watermark `w`, flushes the transform's internal state, and may emit derived records and watermarks. `ProcessWm(w)` is always invoked only after `ProcessRecord(r)` consumes all records in the current epoch.

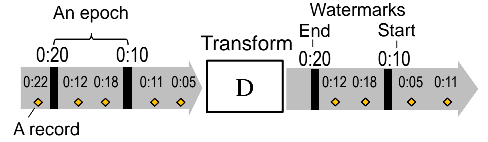
3 Design goals and criteria

We seek to exploit the potential of modern multicore hardware with its abundant hardware parallelism, memory capacity, and I/O bandwidth for high throughput and low latency. A key contribution of this paper is exploiting *epoch parallelism* by concurrently processing all available epochs in every transform, in addition to pipeline parallelism. Epoch parallelism generalizes the idea of processing the records in each epoch out-of-order by processing epochs out-of-order. The following two invariants ensure correctness:

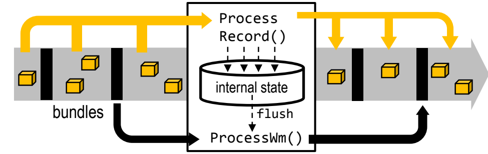
(1) *Records respect epoch boundaries* Each epoch is defined by a start watermark w_{start} and an end watermark w_{end} that arrive at ingress at time *start* and *end*, and consists only of records r_{at} that arrive at ingress at time *at*, with $start < at < end$. Once an ingress record r_{at} is assigned an epoch, records never changed epochs, since this change might violate the watermark guarantee.

(2) *Watermark ordering* A transform D may only consume w_{end} after it consumes all the records r in the epoch. This invariant transitively ensures that watermarks and epochs are processed in order, and is critical to pipeline correctness, as it enforces the progression contract on ingress and between transforms.

Our primary design goal is to minimize latency by exploiting epoch and pipeline parallelism with minimal synchronization while maintaining these invariants. In particular, our engine processes unconsumed records using all available hardware resources regardless of record ordering, delayed watermarks, or epoch ordering. We further minimize latency by exploiting the multicore memory hierarchy (i) by creating sequential memory layout and minimizing data movement, and (ii) by mapping streaming data flows to the NUMA architecture.



(a) A logic view of a transform, which consumes and produces out-of-order records as delimited by watermarks



(b) A transform in StreamBox. It is executed to consume and produce records belonging to multiple epochs in parallel.

Figure 1: A transform in a StreamBox pipeline.

4 StreamBox overview

A StreamBox pipeline includes multiple transforms and each transform has multiple containers. Each container is linked to a container in a downstream transform or egress. Containers form a network pipeline organization, as depicted in Figure 2. Records, derived records, and watermarks flow through the network by following the links. A window consists of one or more epochs. The window size determines the output aggregation and memory layout, but otherwise does not influence how StreamBox manages epochs.

This dataflow pipeline network is necessary to exploit parallelism because parallelism emerges dynamically as a result of variation in record arrival times and the variation in processing times of individual records and watermarks for different transforms. For instance, records, based on their content, may require variable amounts of processing. Furthermore, it is typically faster to process a record than a watermark. However, exposing this abundant record processing parallelism and achieving low latency require prioritizing containers on the critical path through the network. StreamBox prioritizes records in containers with timestamps preceding the pipeline's upcoming output watermark. Otherwise, the scheduler processes records from transforms with the most open containers. StreamBox thus dynamically adds parallelism to the bottleneck transforms of the network to optimize latency.

StreamBox implements three core components:

Elastic pipeline execution StreamBox dynamically allocates worker threads (*evaluators*) from a pool to transforms to maximize CPU utilization. StreamBox pins each evaluator to a CPU core to limit contention. During execution, StreamBox dispatches pending records and watermarks to evaluators. An evaluator executes transform code (i.e., `ProcessRecord()` or `ProcessWm()`) and

produces new records and watermarks that further drive the execution of downstream transforms.

When dispatching records, StreamBox packs them into variable sized *bundles* for processing to amortize dispatch overhead and improve throughput. Bundles differ from batches in many other streaming engines [38, 32, 7]. First, bundle size is completely orthogonal to the transform logic and its windowing scheme. StreamBox is thus at liberty to vary bundle size dynamically per transform, trading dispatch latency for overhead. Second, dynamically packing records in bundles does not delay evaluators and imposes little buffering delay. StreamBox only produces sizable bundles when downstream transforms back up the pipeline.

Cascading containers Each container belongs to a transform and tracks one epoch, its state (*open, processing,* or *consumed*), the relationship between the epoch's records and its end watermark, and the output epoch(s) in the downstream consuming transform(s). Each transform owns a set of containers for its current input epochs. With this container state, executers may concurrently consume and produce records in *all* epochs without breaking or relaxing watermarks.

Pipeline state management StreamBox places records belonging to the same temporal windows (one or more adjacent epochs) in contiguous memory chunks. It adapts a bundle's internal organization of records, catering to data properties, e.g., the number of values per key. StreamBox steers bundles so that they flow mostly within their own NUMA nodes rather than across nodes. To manage transform internal state, StreamBox instantiates a contiguous array of *slides* per transform, where each slide holds processing results for a given event-time range, e.g., a window. Evaluators operate on slide arrays based on window semantics, which are independent of the epoch tracking mechanism – cascading containers. The slide array realization incurs low synchronization costs under concurrent access.

5 Cascading containers

Cascading containers track epochs and orchestrate concurrent evaluators (i) to consume all of an epoch's records before processing its end watermark, (ii) to consume watermarks in stream order, and (iii) to emit records derived from an upstream epoch into the corresponding downstream epoch(s).

Figure 2 shows the cascading container design. Each transform owns a set of input stream containers, one for each potential epoch. When StreamBox creates a container *uc*, it creates one downstream container *dc* (or more) for its output in the downstream transform(s) and links to it, causing a cascade of container creation. It puts all records and watermarks derived from the transform on *uc* into this corresponding downstream container *dc*. All these containers form a pipeline network. As stream processing progresses, the network topology evolves. Evaluators create new containers, establish links between containers, and destroy consumed containers.

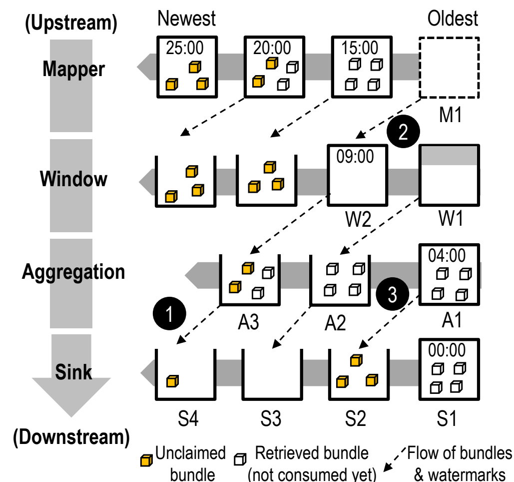


Figure 2: An overview of cascading containers

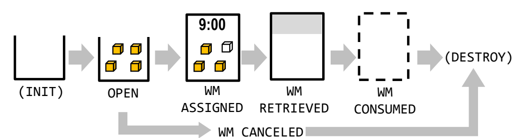


Figure 3: The life cycle of a container

5.1 Container implementation

StreamBox initializes a container D_{own} when the transform receives the first input record or bundle of an epoch. Each container includes any unclaimed bundles of the epoch. An *unconsumed* counter tracks the number of bundles that ever entered the container but are not fully consumed. After processing a bundle, D_{own} deposits derived output bundles in the downstream container and then updates the unconsumed counter.

Container state StreamBox uses a container to track an epoch's life cycle as follows and shown in Figure 3.

OPEN Containers are initially empty. An open container receives bundles from the immediate upstream D_{up}. The owner D_{own} processes the bundles simultaneously.

WM_ASSIGNED When D_{up} emits an epoch watermark w, it deposits w in D_{own}'s dependent container. Eventually D_{own} consumes all bundles in the container and the *unconsumed* counter drops to zero, at which point D_{own} retrieves and processes the end watermark.

WM_RETRIEVED A container enters this state when D_{own} starts processing the end watermark.

WM_CONSUMED After D_{own} consumes the end watermark, it guarantees that it has flushed all derived state and the end watermark to the downstream container and D_{own} may be destroyed.

WM_CANCELLED D_{up} chooses not to emit the end watermark for the (potential) epoch. Section 5.2 describes how we support windowing transforms by cancelling watermarks and merging containers.

Lock-free container processing Containers are lock-free to minimize synchronization overhead. We instantiate the end watermark as an atomic variable that enforces acquire-release memory order. It ensures that D_{own} observes all D_{up} evaluators' writes to the container's unclaimed bundle set before observing D_{up}'s write of the end watermark. The unclaimed bundle set is a concurrent data structure that aggressively weakens the ordering semantics on bundles for scalability. Examples of other such data structures include non-linearizable lock-free queues [13] and relaxed priority queues [4]. We further exploit this flexibility to make the bundle set NUMA-aware, as discussed in Section 7.1.

5.2 Single-input transforms

If a transform has only one input stream, all its input epochs – and therefore the containers – are ordered, even though records are not.

Creating containers The immediate upstream container D_{up} creates downstream containers on-demand and links to them. Figure 2 ❶ shows an example of container creation. When StreamBox processes the first bundle in A3, it creates S4 and any missing container that precedes it, in this case S3, and links A3 to S4 and A2 to S3. To make concurrent growth safe, StreamBox instantiates downstream links and containers using an atomic variable with strong consistency. Subsequent bundle processing uses the instantiated links and containers.

Processing To select a bundle to process, evaluators walk the container list for a transform, starting from the oldest container to the youngest, since the oldest container holds the most urgent work for state externalization. If an evaluator encounters containers in the `wm_consumed` state, it destroys the container. Otherwise,

1. it retrieves an unclaimed bundle. If none exists,
2. it retrieves the end watermark when (i) the watermark is valid (i.e., the container has `wm_assigned`), and (ii) all bundles are consumed (*unconsumed* == 0), and (iii) all watermarks in the preceding containers of D_{own} are consumed.
3. If the evaluator fails to retrieve a bundle or watermark from this container, it moves to the next younger container on D_{own}'s list.

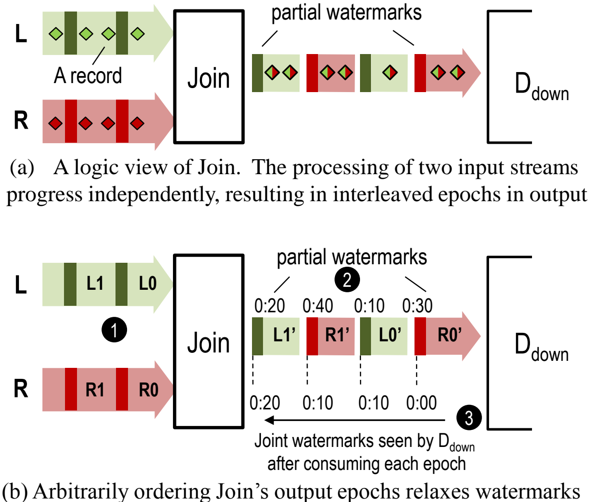


Figure 4: A logic diagram of OOP temporal join

Figure 2 shows an example. An evaluator starts from the oldest container W1 to find work ❷. Because W1 is in WM_RETRIEVED (all bundles are consumed and the end watermark is being processed), the worker moves on to W2. Because all bundles in W2 are consumed but the end watermark is available, it retrieves the watermark (09:00) for processing. Section 6 describes how we prioritize transforms in the container network.

Merging containers for windowing For each input container, we create a *potential* downstream container, expecting each input epoch will correspond to an output epoch. However, when a transform D performs windowing operations, it often must wait for multiple watermarks to correctly aggregate records. In this case, we merge containers. Figure 2 ❸ shows an example of Aggregation on a 10-min window. After consuming container A1 with its 04:00 watermark, the Aggregation transform cannot yet emit a watermark and retire its current window (0:00-10:00). Our solution is to cancel watermarks and merge the downstream output containers until the windowing logic, which uses event time, is satisfied. This operation is cheap. StreamBox cancels watermarks by simply marking them w_{cancel}. As evaluators walk container lists and observe w_{cancel}, they logically treat adjacent containers as one, e.g., S2 and S3. When the transform receives a watermark $ts \geq 10$, it emits the watermark which will eventually close the container.

5.3 Multi-input transforms

A multi-input transform, such as temporal Join and Union, takes multiple input streams and produces one stream. Figure 4 shows an example of out-of-order temporal join [27]. The left and right input streams progress independently (they share D_{join}'s internal state). The

output stream consists of *interleaved* epochs resulting from processing either input stream. These epochs are delimited by *partial* watermarks (w_L or w_R), which are also solely derived from the input streams. The downstream D_{down} derives a *joint* watermark as $min(w'_L, w'_R)$, where w'_L and w'_R are the most recent left and right partial watermarks.

The case for unordered containers A multi-input transform, unlike single-input transforms, cannot always have its downstream containers arranged on an *ordered* list (§5.2) because an optimal ordering of output epochs depends on their respective end (partial) watermarks. On the other hand, arbitrarily ordering output epochs may unnecessarily relax watermarks and delay watermark processing (§2).

Figure 4(b) shows an example of arbitrarily ordering output epochs. While processing *open* input epochs L0/L1 and R0/R1 ❶, StreamBox arbitrarily orders the corresponding output as L1'→R1'→L0'→R0' without knowing the end watermarks. Later, these output epochs eventually receive their *partial* end watermarks ❷. Upon consuming them, D_{down} derives joint watermarks based on its subsequent observations of partial watermarks ❸. Unfortunately, the joint watermark is more relaxed than the partial watermarks. For instance, the partial watermark 00:30 of R0' guarantees that all records in R0' are later than 00:30. However, from the derived joint watermark, D_{down} only knows that they are later than 00:00. Relaxed watermarks propagate to all downstream transforms. To tighten a joint watermark, StreamBox should have placed L0' and L1' (and perhaps more subsequent left epochs) before R0' and R1'. However, it cannot make that decision before observing all these partial watermarks!

In summary, StreamBox must achieve two objectives in tracking epochs for multi-input transforms. (1) It must track output epochs with corresponding containers for epoch parallelism. (2) It must defer ordering these containers until it determines their end watermarks.

Solution StreamBox maintains *unordered* containers for a multi-input transform's output epochs and their downstream counterparts. Once StreamBox determines the ordering of one epoch, it appends the corresponding container to an ordered list and propagates this change downstream. Figure 5 shows an example.

- D_{join} owns two ordered container lists L and R.
- D_1, the immediate downstream transform of D_{join}, owns three ordered lists of containers. L1 and R1 are derived from D_{join}'s L and R, respectively. S1 holds merged containers from L1 and R1.
- With D_2 downstream of D_1, D_2 owns an unordered set U and an ordered list S2.

As D_{join} processes its input streams L and R, it deposits the derived bundles and watermarks to containers on L1, R1, and S1 ❶. D_1 selects the oldest container C1 on L1 and R1 to process and it appends C1 to S1 ❷. Processing C1, deposits records in container C2 (following the *down* link), which subsequently produces records in containers at S2 ❸ and beyond ❹.

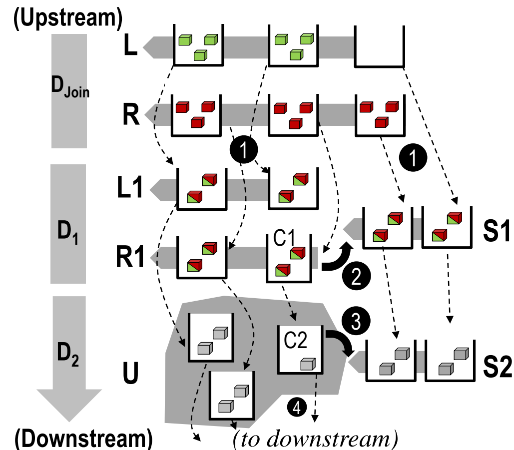


Figure 5: Unordered containers for Join and its downstream. For brevity, container watermarks are not drawn

5.4 Synchronized access to containers

In the cascading containers network, the concurrent evaluators dynamically modify the network topology by creating, linking, and destroying containers. Although the most frequent container operations, such as processing records, are lock-free as described in Section 5.1, modifying the container network must be synchronized. We carefully structure network modifications in reader and writer paths and synchronize them with one readers-writer lock for each container list. To retrieve work, an evaluator holds the container list's reader lock while walking the list. If the evaluator needs to modify the list (e.g., to destroy a container), it atomically upgrades the reader lock to a writer lock.

6 Pipeline scheduling

A pipeline's latency depends on how fast the engine externalizes the state of the current window. To this end, StreamBox's scheduler prioritizes upcoming state externalization.

StreamBox maintains a global notion of the *next externalization moment (NEM)*. The upcoming windowed output requires processing of all bundles and watermarks with timestamps prior to NEM. After each state externalization, StreamBox increments the NEM monotonically based on a prediction. In the common case where externalization is driven by temporal windows, the engine can

accurately predict NEM as the end of the current window. In case windowing information is unavailable, the engine may predict NEM based on historical externalization timing. Mispredicting NEM may increase the output delay but will not affect correctness.

NEM guides work prioritization in StreamBox. All evaluators independently retrieve work (i.e., bundles or watermarks) from cascading containers. By executing StreamBox's dispatch function, an evaluator looks for work by traversing container lists from the oldest to the youngest, starting from the top of the network. It prioritizes bundles in containers with timestamps that precede NEM.

Watermark processing is on the critical path of state externalization and often entails substantial amount of work, e.g., reduction of the window state. To accelerate watermark processing, StreamBox creates a special *watermark task queue*. Watermark tasks are defined as lambda functions. StreamBox gives these tasks higher priority and executes them with the same set of evaluators – without oversubscribing the CPU cores. An evaluator first processes watermark tasks. After completing a task, evaluators return to the dispatcher immediately. Evaluators never wait on a synchronization barrier inside the watermark evaluator. This on-demand, extra parallelism accelerates watermark evaluation.

7 Pipeline state management

The memory behavior of a stream pipeline is determined by the bundles of records flowing among transforms and the transforms' internal states. To manage this state, StreamBox targets locality, NUMA-awareness, and coarse-grained allocation/free. We decouple state management from other key aspects, including epoch tracking, worker scheduling, and transform logic.

7.1 Bundles

Adaptive internal structure StreamBox adaptively packs records into bundles for processing.

StreamBox seeks to (i) maximize sequential access, (ii) minimize data movement, and (iii) minimize the *per-record* overhead incurred by bundling.

A bundle stores a "flat" sequence of records sequentially in contiguous memory chunks. This logical record ordering supports grouping records temporally in epochs and windows, and by keys. It achieves both because temporal computation usually executes on all the keys of specific windows, rather than on specific keys of all windows. This choice contrasts to prior work that simply treats <window, key> as a new key.

To minimize data movement, StreamBox adapts bundle internals to the transform algorithm. For instance, given a Mapper that filters records, the bundles include both records and a bitmap, where each bit indicates the presence of a record, so that a record can be logically filtered by simply toggling a bit. Databases commonly use this optimization [7] as well.

StreamBox adapts bundle internals based on input data properties. The performance of keyed transforms, i.e., those consuming key-value pairs, is sensitive to the physical organization of these values. If each key has a large number of values, a bundle will hold a key's values using an array of pointers, each pointing to an array of values. This choice makes combining values produced by multiple workers as cheap as copying a few pointers. If each key only has a few values, StreamBox holds them in an array and copies them during combining. To learn about the input data, StreamBox samples a small fraction of it.

NUMA-aware bundle flows StreamBox explicitly steers bundles between transforms for NUMA locality by maximizing the chance that a bundle is both produced and consumed on the same NUMA node.

Each bundle resides in memory from one NUMA node and is labeled with that node. When an evaluator processes a container, it prefers unclaimed bundles labeled with its same NUMA node. It will process non-local bundles only when bundles from the local node are all consumed. To facilitate this process, an evaluator always allocates memory on its NUMA node, and later deposits the new bundle to the NUMA node of the downstream container. Notice that the NUMA-aware scheduling only affects the order among bundles within a container. It does not starve important work, e.g., containers to be dispatched by the next externalization moment.

7.2 Transform Internal State

StreamBox organizes a transform's internal state as an array of temporal *slides*, forming a slide. Each slide corresponds to a window (for fixed windows) or a window's offset (for sliding windows). Note that the size of a slide is independent of an epoch size.

To access a transform's state, an evaluator operates on a range of slides: updating slides in-place for accumulating processing results; fetching slides for closing a window; and retiring slides for state flushing. Since concurrent evaluators frequently access the slide arrays, we need to minimize locking and data movement. To achieve this goal, StreamBox grows the array on-demand and atomically. It only copies pointers when fetching slides. It decouples the logical retirement of slides from their actual, likely expensive destruction. To support concurrent access to a single slide, the current StreamBox implementation employs off-the-shelf concurrent data structures, as discussed below.

56CM *Dell PowerEdge R930*	
4x14 Xeon E7-4850v4 “Broadwell”, 256GB DRAM, Linux 4.4	
12CM *Dell PowerEdge R720*	
2x6 Xeon E5-2630v2 “Ivy Bridge”, 256GB DRAM, Linux 4.4	

Table 2: Test platforms used in experiments

8 Implementation

We implement StreamBox in 22K SLoC of C++11. The implementation extensively uses templates, static polymorphism, and C++ smart pointers. We implemented Windowing, GroupBy, Aggregation, Mapper, Reducer, and Temporal Join as our library transforms. Our scalable parallel runtime relies on the following scalable low-level building blocks.

C++ libraries We use boost [20] for timekeeping and locks, Intel TBB [22] for concurrent hash tables, and Facebook folly [17] for optimized vectors and strings. Folly improves the performance of some benchmarks by 20–30%. For scalable memory allocation, we use `jemalloc` [12], which scales much better than `std::alloc` and TBB [23] on our workloads.

Concurrent hash tables are hotspots in most statefull pipelines. We tested three open-source concurrent hash tables [22, 18, 17], but they either did not scale to a large core count or required pre-allocating a large amount of memory. Despite the extensive research on scalable hash tables [26, 6], we needed to implement an internally partitioned hash table. We wrapped TBB's concurrent hash map. This simple optimization improves our performance by 20–30%.

Bundle size is an empirical trade off between scheduling delay and overhead. StreamBox mainly varies bundle size at pipeline ingress. When the engine is fully busy, with all records in one ingress epoch, it produces as many bundles as evaluators, e.g., 56 bundles for 56 evaluators, to maximize the bundle size without starving any thread. The largest bundle size is around 80K records. When the ingress slows down, the system shrinks bundle sizes to reduce latency. We empirically determine that a 2$\times$ reduction in bundle size balances a 10% drop in ingress data rate. We set the minimal bundle size at 1K records to avoid excessive per-record overhead.

9 Evaluation

Methodology We evaluate StreamBox on the two multicore servers, summarized in Table 2. 56CM is a high-end server that excels at real-time analytics and 12CM is a mid-range server. Although 100 Gb/s Infiniband (RDMA) networks are available, our local network is only 10 Gb/s. However, 10 Gb/s is insufficient to test StreamBox and furthermore even if we used Infiniband, it will directly store stream input in memory. We therefore generate ingress streams from memory. We dedicate a small number of cores (1–3) to the pipeline source. We then replay these large memory buffers pre-populated with records and emit in-memory stream epochs continuously. We measure the maximum sustained throughput of up to 38 GB/s at the pipeline source when the pipeline delay meets a given target.

Benchmarks We use the following benchmarks and datasets. Unless stated otherwise, each input epoch contains 1 M records and spans 1 second of event time. (1) **Windowed Grep (grep)** searches the input text and outputs all occurrences of a specific string. We use Amazon Movie Reviews (8.7 GB in total) [37] as input, a sliding window of 30 seconds, and 1 second target latency. The input record size is 1 KB. (2) **Word Count (wordcount)** splits input texts into words and counts the occurrences of each word. We use 954 MB English books [21] as input, a sliding window of 30 seconds, and 1 second target latency. The input record size is 100 bytes. (3) **Temporal Join (join)** has two input streams, for which we randomly generate unique 64-bit integers as keys. The join window for each record is $\pm$ 0.5 seconds. (4) **Counting Distinct URLs (distinct)** [32] counts unique URL identifiers. We use the `Yandex` dataset [16] with 70 M unique URLs and a fixed window of 1 second. (5) **Network Latency Monitoring (netmon)** [32] groups network latency records by IP pairs and computes the average per group. We use the Pingmesh dataset [19] with 88 M records and a fixed window of 1 second. The source emits 500K records per epoch. (6) **Tweets Sentiment Analysis (tweets)** [32] correlates sentiment changes in a tweet stream to the most frequent words. It correlates results from two pipelines: one that selects windows with significant sentiment score changes, and the other that calculates the most frequent words for each window. We use a public dataset of 8 million English tweets [10] and a fixed window of 1 second. This benchmark is the most complex and uses 8 transforms.

9.1 Throughput and Scalability

This section evaluates the throughput, scalability, and out-of-order handling of StreamBox, and compares with existing stream processing systems.

Throughput Figure 6 presents throughput on the y-axis for the six benchmarks as a function of hardware parallelism on the x-axis and latency as distinct lines. StreamBox has high throughput and typically processes millions of input records per second on a single machine, while delivering latencies as low as 50 ms. In particular, `grep` achieves up to 38 M records per second, which translates to 38 GB per second. This outstanding performance is due to low overheads and high system utilization. Profil-

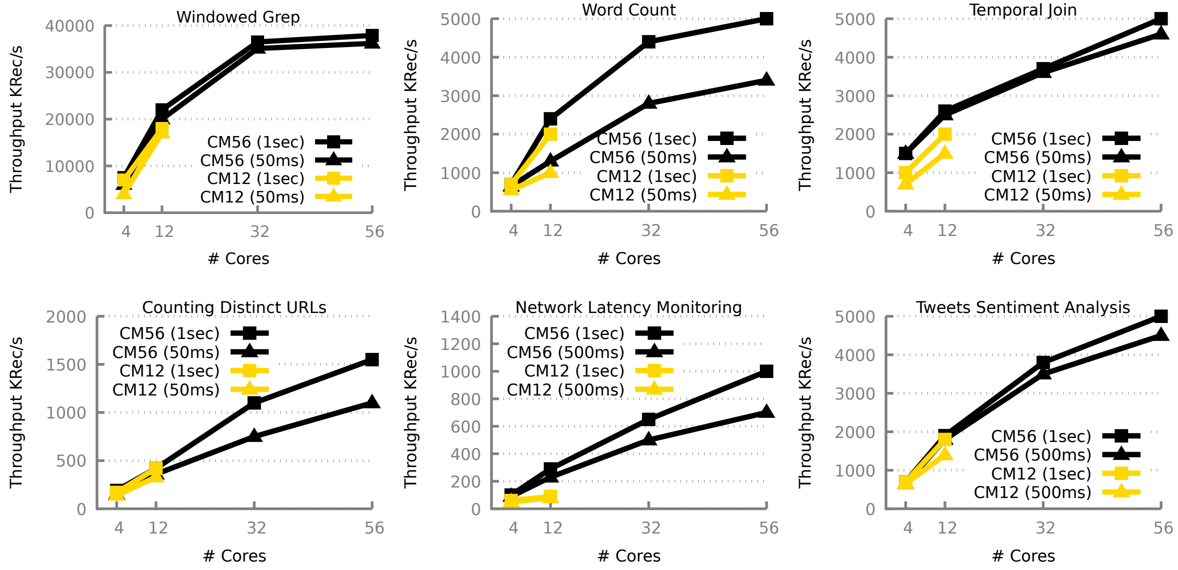
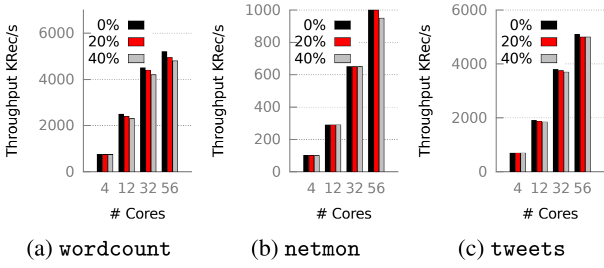


Figure 6: Throughput of StreamBox as a function of hardware parallelism and latency. StreamBox scales well.



(a) `wordcount` (b) `netmon` (c) `tweets`

Figure 7: StreamBox achieves high throughput even when a large fraction of records arrive out-of-order.

ing shows that all CPU cores have consistently high utilization ($> 95\%$) and that most time is spent performing transform logic, e.g., processing stream data and manipulating hash tables.

Scalability Figure 6 shows that StreamBox scales well with core count for most benchmarks on both the 12-core and 56-core machines. When scalability diminishes in a few cases beyond 32 cores, as for `grep`, it is a result of memory-bound computation saturating the machine.

Out-of-order records By design, StreamBox efficiently computes on out-of-order records. To demonstrate this feature, we force a certain percent of records to arrive early in each epoch, i.e., the event time of these records is larger than the enclosing epoch's end watermark. Figure 7 shows the effect on throughput for 3 benchmarks. StreamBox achieves nearly the same throughput and latency as in in-order data processing. In particular, the throughput loss is as small as 7% even with 40% of records out-of-order. The minor degradation is due to early-arriving records that accumulate more windows in the pipeline. We attribute this consistent performance to (i) out-of-order epoch processing, since each transform continuously processes out-of-order records without delay, and (ii) prioritizing bundles and watermarks that decide the externalization latency of the current window in the scheduler.

Comparing to distributed stream engines We first compare StreamBox with published results of a few popular distributed stream processing systems and then evaluate two of them on our 56-core machine. Most published results are based on processing of in-order stream data. For out-of-order data, they either lack support (e.g., no notion of watermarks) or expect transforms to "hold and sort", which significantly degrades latency [11, 35].

Compared to existing systems, StreamBox jointly achieves low millisecond latency and high throughput (tens of millions of records per second). Very few systems achieve both. To achieve similar throughput, prior work uses at least a medium-size cluster with a few hundred CPU cores [28, 38]. For instance, under the 50-ms target latency, StreamBox's throughput on 56CM is 40$\times$ greater than StreamScope [28] running on 100 cores. Moreover, even under a 1-second target latency, StreamBox achieves much higher throughput per core. StreamBox can process 700K records/sec for `grep` and 90K records/sec for `wordcount` per core, which are 4.7$\times$ and 1.5$\times$ faster than the per-core processing rate reported by Spark Streaming on a 100-node cluster with a total of 400 cores.

We further experimentally compare StreamBox with Spark (v2.1.0) [38] and Apache Beam (v0.5.0) [3], on the same machine (56CM). We verify that they both utilize all cores. We set the the target latency to 1 second since they cannot achieve 50 ms as StreamBox does. Fig-

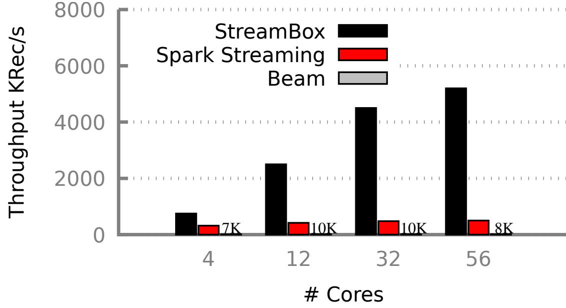


Figure 8: StreamBox scales better than Spark and Beam with `wordcount` on 56CM, with a 1-second target latency.

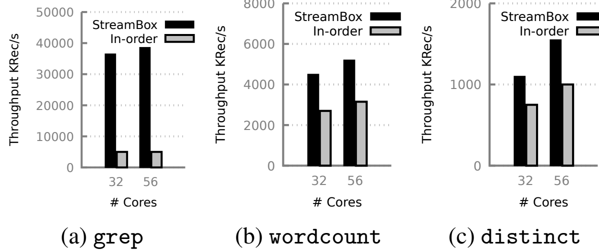


Figure 9: In-order processing reduces parallelism, scalability, and throughput.

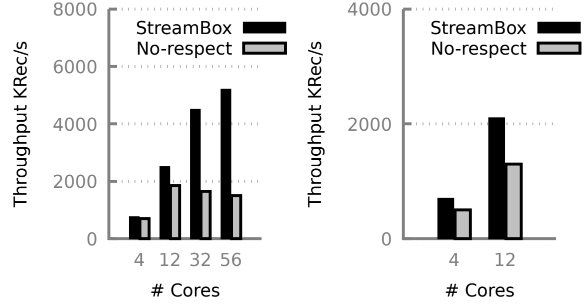


Figure 10: When records do not respect epoch boundaries, it limits parallelism, scalability, and throughput.

ure 8 shows that StreamBox achieves significantly higher throughput (by more than one order of magnitude) and it scales much better with core count.

Comparing to single-machine streaming engines A few streaming engines are designed for a single machine: Oracle CEP [31], StreamBase [34], Esper [14], and SABER (for CPU+GPU) [24]. With 4 to 16 CPU cores, they achieve throughput between thousands and a few million of records per second. None of them reports to scale beyond 32 CPU cores. In particular, we tested Esper [14] on 56CM with `wordcount`. On four cores, Esper achieves around 900K records per second, which is similar to StreamBox with the same core count. However, we were unable to get Esper to scale even after applying recommended programming techniques, e.g., context partitioning [15]. As the core count increases, we observed the throughput drops.

In summary, StreamBox achieves better or similar per core performance than prior work. More importantly, StreamBox scales well to a large core count even with out-of-order record arrival.

9.2 Validation of key design features

This section evaluates the performance and scaling contributions of our key design features.

Epoch parallelism for out-of-order processing Epoch parallelism is fundamental to producing abundant parallelism and exploiting out-of-order processing. We compare with in-order epoch processing by implementing "hold and sort," in which each transform waits to process an epoch until all its records arrive. Note that this in-order epoch processing leaves out the high cost of sorting records. It processes records within an epoch out-of-order. Figure 9 shows that in-order epoch processing reduces throughput by 25% – 87%. Profiling reveals the reduced parallelism causes poor CPU utilization.

Records must respect epoch boundaries (§3). StreamBox enforces the invariant that records respect epoch boundaries by mapping upstream containers to downstream containers (§5). We compare this to an alternative design where a transform's output records always flow into the *most recently* opened downstream container. Records then no longer respect epoch boundaries, since later records may enter earlier epochs. Violating the epoch invariant leads to huge latency fluctuations in watermark externalization, degrading performance. Figure 10 shows that not respecting epoch boundaries reduces throughput by up to 71%.

Prioritized scheduling (§6) Prioritizing containers on the critical path is crucial to latency and throughput. To explore its effect, we disable prioritized scheduling such that evaluators freely retrieve available bundles anywhere in the pipeline starting from its curent source and sink container. In this configuration, evaluators tend to rush into one transform, drain bundles there, and then move to the next. We confirmed this behavior with profiling. Performance measurements show that the pipeline latency fluctuates greatly and sometimes overshoots the target latency by a factor of 10.

NUMA-awareness (§7) We find NUMA-awareness especially benefits memory-bound benchmarks. For example, grep without windowing achieves 54 GB/s on 56CM, which is 12.5% higher than a configuration with NUMA-unaware evaluators.

Watermark arrival rates. Frequent watermarks lead to shorter epochs and more containers, each with fewer records, thus increasing the maintenance cost of cascad-

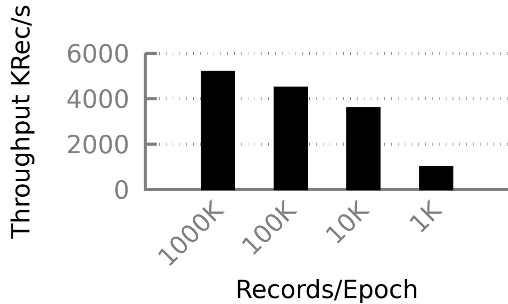


Figure 11: Performance impact of watermark arrival rate for `wordcount` on 56CM.

ing containers. In general, as shown in Figure 11, containers are sufficiently lightweight so that frequent watermarks (e.g., 100× more watermarks in 10K records/epoch) result in only a minor performance loss (e.g., 20%). However, substantial performance degradation emerges for watermarks at the rate of 1 K records/epoch, because frequent container creation and destruction incur too much synchronization.

10 Related work

This section compares StreamBox to prior work that uses the out-of-order processing (OOP) model, distributed and single server stream engines, and on exploiting shared memory for streaming.

OOP stream processing A variety of classic streaming engines focus on processing in-order records with a single core (e.g., StreamBase [34], Aurora [1], TelegraphCQ [9], Esper [14], Gigascope [11], and NiagaraST [29]). Li et al. [27] advocate OOP stream processing that relies on stream progression messages, e.g. punctuations, for better throughput and efficiency. The notion of punctuations is implemented in many modern streaming engines [3, 7, 28]. These systems do exploit pipeline and batch parallelism, but they do not exploit out-of-order processing of epochs to expose and deliver highly scalable data parallelism on a single server.

Single-machine streaming engines Trill [8] inspires StreamBox's state management with its columnar store and bit-vector design. However, Trill's punctuations are generated by the engine itself in order to flush its internal batches, which limits parallelism. Furthermore, Trill assumes ordered input records, which limits it applicability. StreamBox has neither of these limitations. SABER [24] is a hybrid streaming engine for CPUs and GPGPUs. Similar to StreamBox, it exploits data parallelism with multithreading. However, SABER does not support OOP. It must reorder execution results from concurrent workers, limiting its applicability and scalability. Oracle CEP [31] exploits record parallelism by relaxing record ordering. However, it lacks the notion of watermarks and does not implement statefull OOP pipelines.

Distributed streaming engines Several systems process large continuous streams using hundreds to thousands of machines. Their designs often focus on addressing the pressing concerns of a distributed environment, such as fault tolerance [38, 32, 28], programming models [30, 3], and API compatibility [36]. TimeStream [32] tracks data dependence between transform's input and output, but uses it for failure recovery. StreamBox also tracks fine-grained epoch dependences, but for minimizing externalization latency. StreamScope [28] handles OOP using watermark semantics, but it does not exploit OOP for performance as does StreamBox. It instead implements operator determinism based on holding and waiting for watermarks. StreamBox is partially inspired by Google's dataflow model [3] and is an implementation of its OOP programming model. However, to the best of our knowledge and based on our experiments, the Apache Beam [5] open-source implementation of Google dataflow does not exploit epoch parallelism on a multicore machine.

Data analytics on a shared memory machine Some data analytics engines propose to facilitate sequential memory access [33, 25] and one exploits NUMA [39]. StreamBox's bundles are similar to morsels in a relational query evaluator design [26], where evaluators process data fragments ("morsels") in batch and that are likely allocated on local NUMA nodes. StreamBox favors low scheduling delay for stream processing. Evaluators are rescheduled after consuming each bundle, instead of executing the entire pipeline for that bundle.

11 Conclusions

This paper presents the design of a stream processing engine that harnesses the hardware parallelism and memory hierarchy of modern multicore servers. We introduce a novel data structure called *cascading containers* to track dependences between epochs while at the same time processing any available records in any epoch. Experimental results show StreamBox scales to a large number of cores and achieves throughput on-par with distributed engines on medium-size clusters. At the same time, StreamBox delivers latencies in the tens of milliseconds, which are 20× shorter than other large-scale streaming engines. The key contribution of our work is a generalization of out-of-order record processing to out-of-order epoch processing that maximizes parallelism while minimizing synchronization overheads.

Acknowledgments

This work was supported in part by NSF Award #1619075 and by a Google Faculty Award. The authors thank the anonymous reviewers and the paper shepherd, Charlie Curtsinger, for their useful feedback.

References

[1] ABADI, D., CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., ERWIN, C., GALVEZ, E., HATOUN, M., MASKEY, A., RASIN, A., ET AL. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 666–666.

[2] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1033–1044.

[3] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., ET AL. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1792–1803.

[4] ALISTARH, D., KOPINSKY, J., LI, J., AND SHAVIT, N. The spraylist: A scalable relaxed priority queue. *SIGPLAN Not. 50*, 8 (Jan. 2015), 11–20.

[5] APACHE. Beam. `https://beam.apache.org/`, 2017.

[6] BALKESEN, C., TEUBNER, J., ALONSO, G., AND ÖZSU, M. T. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (2013), IEEE, pp. 362–373.

[7] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment 8*, 4 (2014), 401–412.

[8] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment 8*, 4 (2014), 401–412.

[9] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., REISS, F., AND SHAH, M. A. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 668–668.

[10] CHENG, Z., CAVERLEE, J., AND LEE, K. You are where you tweet: a content-based approach to geo-locating twitter users. In *Proceedings of the 19th ACM international conference on Information and knowledge management* (2010), ACM, pp. 759–768.

[11] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 647–651.

[12] DAVID GOLDBLATT, DAVE WATSON, J. E. Jemalloc memory allocator. `http://http://jemalloc.net/`, 2017.

[13] DESROCHERS, C. moodycamel::concurrentqueue. `https://github.com/cameron314/concurrentqueue`, 2016.

[14] ESPERTECH. Esper. `http://www.espertech.com/esper/`, 2017.

[15] ESPERTECH. Esper faq. `http://www.espertech.com/esper/faq_esper.php#scaling`, 2017.

[16] EUGENE KHARITONOV, P. S. Yandex: Personalized web search challenge. `https://www.kaggle.com/c/yandex-personalized-web-search-challenge/data`, 2017.

[17] FACEBOOK. Folly. `https://github.com/facebook/folly#folly-facebook-open-source-library`, 2017.

[18] GOYAL, M., FAN, B., LI, X., ANDERSEN, D. G., AND KAMINSKY, M. Libcuckoo. `https://github.com/efficient/libcuckoo`, 2017.

[19] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 139–152.

[20] GURTOVOYI, A., AND ABRAHAMSI, D. Boost c++ libraries. http://www.boost.org/, 2017.

[21] HART, M. Free ebooks by project gutenberg. http://www.gutenberg.org/wiki/Main_Page, 2017.

[22] INTEL. Intel threading building blocks. https://software.intel.com/en-us/intel-tbb, 2017.

[23] INTEL. Scalable memory allocator. https://www.threadingbuildingblocks.org/tutorial-intel-tbb-scalable-memory-allocator, 2017.

[24] KOLIOUSIS, A., WEIDLICH, M., CASTRO FERNANDEZ, R., WOLF, A. L., COSTA, P., AND PIETZUCH, P. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 555–569.

[25] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[26] LEIS, V., BONCZ, P., KEMPER, A., AND NEUMANN, T. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 743–754.

[27] LI, J., TUFTE, K., SHKAPENYUK, V., PAPADIMOS, V., JOHNSON, T., AND MAIER, D. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment 1*, 1 (2008), 274–288.

[28] LIN, W., QIAN, Z., XU, J., YANG, S., ZHOU, J., AND ZHOU, L. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. of NSDI* (2016), pp. 439–454.

[29] MAIER, D., LI, J., TUCKER, P., TUFTE, K., AND PAPADIMOS, V. Semantics of data streams and operators. In *International Conference on Database Theory* (2005), Springer, pp. 37–52.

[30] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[31] ORACLE. Stream explorer. http://bit.ly/1L6tKz3, 2017.

[32] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., AND ZHANG, Z. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[33] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[34] STANLEY ZDONIK, MICHAEL STONEBRAKER, M. C. Streambase systems. http://www.tibco.com/products/tibco-streambase, 2017.

[35] TUCKER, P. A., MAIER, D., SHEARD, T., AND FEGARAS, L. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering 15*, 3 (2003), 555–568.

[36] TWITTER. Heron. https://twitter.github.io/heron/, 2017.

[37] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ET AL. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014), IEEE, pp. 488–499.

[38] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 423–438.

[39] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. *SIGPLAN Not. 50*, 8 (Jan. 2015), 183–193.