Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow

Tyler Akidau
Snowflake Inc.
Bellevue, WA, USA
takidau@apache.org

Edmon Begoli
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
begolie@ornl.gov

Slava Chernyak
Google Inc.
Seattle, WA, USA
chernyak@google.com

Fabian Hueske
Ververica GmbH
Berlin, Germany
fhueske@apache.org

Kathryn Knight
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
knightke@ornl.gov

Kenneth Knowles
Google Inc.
Seattle, WA, USA
klk@google.com

Daniel Mills
Google Inc.
Seattle, WA, USA
millsd@google.com

Dan Sotolongo
Snowflake Inc.
Bellevue, WA, USA
dan.sotolongo@snowflake.com

ABSTRACT

Streaming data processing is an exercise in taming disorder: from oftentimes huge torrents of information, we hope to extract powerful and timely analyses. But when dealing with streaming data, the unbounded and temporally disordered nature of real-world streams introduces a critical challenge: how does one reason about the completeness of a stream that never ends? In this paper, we present a comprehensive definition and analysis of *watermarks*, a key tool for reasoning about temporal completeness in infinite streams.

First, we describe what watermarks are and why they are important, highlighting how they address a suite of stream processing needs that are poorly served by eventually-consistent approaches:

- Computing a *single* correct answer, as in notifications.
- Reasoning about a *lack* of data, as in dip detection.
- Performing *non-incremental* processing over temporal subsets of an infinite stream, as in statistical anomaly detection with cubic spline models.
- Safely and punctually *garbage collecting* obsolete inputs and intermediate state.
- Surfacing a reliable signal of overall *pipeline health*.

Second, we describe, evaluate, and compare the semantically equivalent, but starkly different, watermark implementations in two modern stream processing engines: Apache Flink and Google Cloud Dataflow.

PVLDB Reference Format:
Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. Watermarks in Stream Processing Systems. PVLDB, 14(12): 3135 - 3147, 2021.
doi:10.14778/3476311.3476389

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://s.apache.org/watermarks-paper-beam-pipeline.

1 INTRODUCTION

By now, the distinction between event time and processing time, the disorder induced by the myriad distributed systems involved in modern data processing pipelines, and the need for intentional approaches to taming that disorder are well recognized [2]. Less understood is one of the key tools for taming disorder: watermarks.

The problem is reliably processing multi-source streams in distributed environments: different producers send events at varying rates, the events flow along distinct paths, encountering varying delays, and arriving at different consumers at different times. How does one efficiently and reliably reason about the completeness of such input data over time?

Flink and Cloud Dataflow address this problem with *watermarks*: quantitative markers associated with a stream that indicate no future events within a stream will have a timestamp earlier than their timestamp.

In Section 2, we provide a definition of watermarks and discuss some of their key uses. In Section 3, we categorize related work across research and industry, and present initial arguments for why we believe watermarks are the best known solution. In Section 4, we explore the semantics of watermarks in detail. In Section 5, we describe the two very different watermark implementations in Flink and Cloud Dataflow. In Section 6, we evaluate those implementations side by side. In Section 7 we briefly discuss future work. And in Section 8, we summarize our takeaways.

2 BACKGROUND

A watermark represents the temporal completeness of an out-of-order data stream. The watermark's current value informs a processor that all messages with a lower timestamp have been received. Watermarks are propagated by a stream processing system from event sources to processors, enabling order-dependent processing on out-of-order streams throughout a data flow graph.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476389

We will more concretely discuss the applications of completeness in Section 2.1, but begin with an example: in a case of online auction processing, a bid processor might wait until it has received all events which occurred before the end time of an auction. At that point, all valid bids for that auction should have arrived and a final winner may be determined, even in a system where events can arrive out-of-order and with substantial delays. Watermarks are a key building block for such logic.

Intuitively, a watermark is a correspondence between processing time and event time (called system time and application time in SQL:2011 [12], or transaction time and valid time in temporal database literature [11]). Processing time is the time at which events are observed or processed by a consumer; event time is the time at which those events occurred in the real world. At every moment in processing time, the watermark yields a timestamp that represents a lower bound of the event times of all *unreceived* events. In other words, if a watermark of t_w is observed, the observer knows that all records with event times less than or equal to t_w have been received, and all future records will have event times greater than t_w.

More formally, for a node N of a dataflow graph G and a totally-ordered time domain T, let I be the sequence of input elements arriving at N. The arrival timing of these elements is given by a processing time function $p : I \rightarrow T$. Each element is tagged with an event timestamp, given by the function $e : I \rightarrow T$. A *watermark* for N is a function $w : T \rightarrow T$ satisfying the following properties.

- **Monotonicity**: w is monotone. It must never "move backwards".
- **Conformance**[1]: for all $i \in I$, $w(p(i)) < e(i)$. A conformant watermark must bound event times from below.
- **Liveness**: $w(t)$ has no upper bound.

These properties require that watermarks eventually make forward progress, but the watermark lag, $t - w(t)$, can be unbounded. Bounding the lag imposes a constraint on the event sequence that the delay of all events be limited to some duration b, such that $\forall i \in I$, $p(i) - e(i) < b$.

This definition is sufficient to provide some intuition about how watermarks function, but elides a number of practical details, particularly concerning watermark generation, propagation, and utilization. We'll discuss these topics in more detail in Section 4, and then describe two implementation approaches in Section 5.

2.1 Key Uses of Completeness

Applications of completeness in stream processing systems typically boil down to managing one of two important aspects of computation over an infinite stream: readiness and obsolescence. Most commonly, we think of readiness and obsolescence in terms of the streaming computation being performed, but they also play a key role in monitoring and understanding data pipeline health over time.

2.1.1 Readiness. For stream processing, readiness refers to the state of having all necessary inputs available for producing some output. While eventually-consistent algorithms are the bread and butter of stream processing, many real-world use cases lend themselves poorly to such approaches where data are processed one at a time, with results continually evolving along the way. For example:

Notifications & alerting: Even when a computation itself is incremental (e.g., SUM, COUNT), the consumer of that incremental computation's output may not support updates, or the partial result being computed along the way may be unstable or misleading until a sufficient amount of input data has been received. A common example is notification and alerting use cases, where the system must generate a single notification containing a single correct answer, not a stream of notifications containing incrementally refined partial results. Watermarks provide a useful completeness signal to generate a single conclusive result based on inputs from a specific range of time, e.g., the COUNT of all login attempts in the last hour.

Data absence: An important category of computation where partial results may be unstable or misleading is use cases which involve reasoning about a *lack* of data, such as dip detection (e.g., detecting when the total count of visits for a time period is *below* an expected threshold). Without a metric of completeness, how is the computation to distinguish between a real dip in event rate and lagging inputs in the event source? Watermarks provide such a metric, making it possible to delay the evaluation of such a predicate until all necessary inputs for the time period in question have been observed.

Non-incremental processing: In other cases, the specific computation being performed may simply be non-incremental. One of the early use cases for MillWheel [1] involved calculating a cubic-spline timeseries model for detecting anomalies in search query patterns. It was not possible to incrementally add new data into this model, so the arrival of additional inputs required fully recomputing the model. At high data volumes, such recomputation can become prohibitively expensive and adds little value. By using watermarks, it is possible to delay computation of the models until all the inputs for a given time window have arrived, after which the model may be computed only once.

Many streaming systems today provide temporal readiness support via watermarks. To name a few, Flink [6] and Dataflow [2] both provide generalized watermarks as a first-class concept in their APIs, while Kafka Streams [17, 22] uses a non-conformant watermark (referred to as a "grace period") to provide final results.

2.1.2 Obsolescence. Obsolescence is the dual of readiness: ensuring that input is not forgotten until all outputs dependent upon it have been computed. Obsolescence is an important part of many streaming pipelines, as it allows for stateful computation over infinite streams without infinite storage.

MillWheel and Cloud Dataflow utilize system-time watermarks to determine when it is safe to garbage collect exactly-once deduplication data on the receiver side of a shuffle between two physical stages in the pipeline [13]. Apache Beam and Apache Flink use watermarks to garbage collect state allocated by a user's computation once no further event can cause it to be observed. Apache Spark's Structured Streaming uses a non-conformant watermark algorithm for garbage collecting intermediate state [8] that is identical to the grace period Kafka Streams uses for its final results feature: track a high watermark of the max event time ever seen within a stream,

[1] Conformant and non-conformant watermarks are sometimes referred to as "perfect" and "heuristic" watermarks, respectively [2, 3].

then offset that by a static allowed-lateness delta to determine the trailing horizon of timestamps which may be discarded.

2.1.3 Health Monitoring. Lastly, it's worth highlighting that a comprehensive and well-instrumented watermark system provides a reliable and general signal of overall pipeline health. Since watermarks track progress throughout a data processing pipeline, any issue that impacts the pipeline's progress will manifest as a delay in its watermarks. Assuming the stream processing framework provides sufficiently fine-grained views of the watermarks (for example, partitioned across and within physical stages in the pipeline), it's often possible to precisely locate an issue in the pipeline by finding the first delayed watermark.

3 RELATED WORK

A survey of research and industry shows there are a variety of approaches to reasoning about completeness. We choose several salient examples from the literature, group them into broad approaches to give a sense of how they work, and order them by increasing generality and complexity.

- **Order-agnostic processing**: The eventually consistent approach to streaming computation [9] sidesteps the complexity of disorder and completeness by ignoring these concerns, but falls short for use cases like those described in Section 2.1 [17]. All streaming systems support this approach to completeness, so we do not discuss it further.
- **Ordered processing**: Processing a collection that is already ordered (usually by time) is a well-established way to guarantee completeness [4, 7, 10, 19, 23]. When each element is processed, the system can assume that no earlier elements remain unprocessed.
- **Watermarks**: Event-time watermarks, the approach we advocate in this paper, are a technique to approximate and react to completeness in time without holding back elements for sorting [1, 2, 6, 8, 18, 22]. Instead, elements are processed out of order, and the watermark indicates an event time before which no elements will enter the system.
- **Timestamp frontiers**: Timestamp frontiers, from the Timely Dataflow model [16], are a generalization of watermarks that allows multiple dimensions of time to advance independently. Timely [14] uses these frontiers as a lightweight coordination mechanism for the parallel execution of nested iterative programs.
- **Punctuations**: Punctuations [20] are a general technique for indicating that no element satisfying a given predicate (such as a timestamp less than a specified t) will appear on a data stream. They subsume the previous approaches, but are difficult to implement generally and scalably.

3.1 Ordered Processing

Many stream processing systems assume events enter the system in order. Examples include complex event processing systems [4, 10, 23] as well as more general-purpose systems like Trill [7]. This ordering assumption is beneficial for multiple reasons. First, such systems can be optimized for performance, in both storage and processing. Second, the processing of the events is computationally deterministic [19], which facilitates testing and debugging. Finally, it gives the system predictable completeness semantics where the arrival of each event guarantees that no earlier event will appear, simplifying the design of in-order systems significantly.

Assuming ordered processing raises practical problems. First, real-world sources often do not provide ordered input. Frequently, events are generated by scattered sources, which makes it challenging to maintain event ordering. The most straightforward solution to this problem is to buffer events until some lateness cutoff threshold, then sort them. The cutoff can be decided in a number of ways, including use of timeouts, estimation, and metadata. However it is decided, the result is a latency penalty for all events as they wait for stragglers to arrive. Second, order is not always needed. Many programs compute order-independent aggregations, like sums, counts, or averages. These aggregations can proceed while waiting for stragglers, creating a substantial opportunity to reduce latency. However, in systems which enforce ordering, all programs are beholden to the delays introduced by the ordering process. Finally, ordering generally requires restrictions on the parallelization of processing, such as a static set of ordered partitions.

3.2 Watermarks

Instead of delaying and then sorting data, watermarks tame out-of-order data by tracking the lowest timestamp that may yet appear in a stream. This approach was originally presented as "heartbeats" by Srivistava and Widom [18], then as "watermarks" in MillWheel [1], Dataflow [2], and Flink [6]. These systems allow events to flow out-of-order, but use a watermark to provide a termination marker. For example, a system finding a *windowed sum* can accumulate the sum until the watermark passes the end of the window, then emit the sum immediately. Because the watermark can adapt to the data, it is not necessary to introduce extra latency to account for stragglers. If incremental semantics are desired, the system can also emit partial aggregates. This allows programmers to take advantage of the latency opportunity offered by stragglers: decrease peak load by spreading out incremental computation while awaiting stragglers.

Unfortunately, out-of-order processing with watermarks brings its own problems. Compared to ordered processing, processing with generalized watermarks is more difficult to implement, reason about, and program for. Debugging and optimizing the resulting programs is yet more subtle. One approach to managing this complexity is to narrow the scope and complexity of the watermark system itself. As noted previously, Kafka Streams [22] and Spark Structured Streaming [8] limit themselves to a simple heuristic watermark for final results and garbage collection, respectively.

Finally, watermarks do not take full advantage of the latency opportunities afforded by some kinds of program. Though watermarks do not delay all input to allow stragglers to arrive, when stragglers do occur in a pipeline they can delay downstream computation arbitrarily even if those computations do not specifically depend on the stragglers.

3.3 Timestamp Frontiers

The Timely Dataflow model [16] generalizes watermarks to track progress along multiple time domains simultaneously. Just like watermark-based systems, Timely systems allow programs to hold the minimum timestamp frontier and register for notification when the frontier passes a timestamp. Each operator maintains a lower bound of the timestamps it is capable of emitting, and the system propagates these bounds between operators. Support for orthogonal time domains allows Timely programs to perform certain tasks, like the simultaneous incremental and iterative processing of Differential Dataflow [15], with unprecedented performance.

Unfortunately, the timestamps that result from working with multiple independent time domains are even more difficult to program than one-dimensional watermarks. Between out-of-order elements, state, frontiers, and capabilities, implementing sophisticated operators in Timely is an intimidating prospect. Fortunately, the library provides many APIs for conveniently building simple operators. This facilitates the simple case, but when complex operators are needed, convenience APIs do not mitigate the complexity described above.

3.4 Punctuations

Punctuations [20] are markers inserted into the stream to specify the end of a subset of data. It is a mechanism for a streaming operator to declare that no more elements will be produced that match some predicate. From this perspective, a watermark value w at some operator is a punctuation declaring that elements with timestamps less than w will not be produced in the future. Similarly, the release of a Timely *capability* c is a punctuation that the releasing operator will not produce elements with timestamp less than c. Since punctuations work with arbitrary predicates, they can be used to signal arbitrary completeness. For example, the original punctuations paper uses them to identify obsolete state in a deduplication operator.

Unfortunately, the generality of punctuations is also their weakness. Each operator needs to implement a complex set of functions for updating and propagating the punctuations through the pipeline, and implementing these is both critical for correctness and difficult. Punctuations also introduce an unfortunate coupling between operators. In a general punctuation-based system, an operator may be expecting predicates over certain properties that upstream operators simply do not implement. Conversely, upstream operators may produce punctuations that downstream operators cannot use, creating inefficiencies. Further theoretical insight is needed to mitigate the programmatic toil imposed by watermarks, timestamp frontiers, and punctuations.

3.5 Why Watermarks?

Of the approaches described above, we believe watermarks provide the best balance of costs and benefits:

- Watermarks allow a system to **react dynamically to changing input disorder**, and do not impose a fixed latency cost as do bounded disorder approaches.
- They **allow incremental processing to proceed** even as the system awaits a signal that inputs are complete.
- A general watermarking system may be **adapted to a variety of different use cases**, with specialized watermark generation logic tailored to capitalize on unique characteristics of a given input source.
- Shortcomings such as stragglers may often be mitigated by **relaxing the completeness constraints** enforced by the system, as appropriate.
- Although the complexity of a generalized watermark implementation is indeed a very real challenge, **much of the complexity burden falls on the *implementation* of the *framework*** itself, not its usage; systems which eschew providing native completeness primitives simply push the complexity of that task on their users. In Section 5, we discuss two different implementations of watermark frameworks.
- Meanwhile, more general approaches such as timestamp frontiers and punctuations bring additional complexity for **arguably diminishing returns**, given that most use cases do not involve iterative computation.

It is for these reasons that we believe watermarks sit in the proverbial sweet spot of cost/benefit tradeoffs for approaches to reasoning about completeness in unbounded stream processing.

4 WATERMARK SEMANTICS

In Section 2, we defined watermarks in the context of a single node in a dataflow graph. In this section, we elaborate on the behavior that emerges from this local definition when interactions across the dataflow graph are considered. We begin by defining the system model and interpreting it in terms of real-world concepts, then discuss the three ways that users of stream processing systems interact with watermarks: *generating* them from time sources, *propagating* them through the program graph, and *consuming* them to determine the completeness of data.

Along the way, we illustrate the concepts we discuss using the example pipeline in Figures 1 and 2. These diagrams show snapshots of the same pipeline taken at two successive points in time, thus allowing us to observe the changes in the system with respect to watermarks and their side effects as time progresses.

The example itself is a simple streaming analytics pipeline for a multi-platform team game, aggregating hourly, per-team scores across two separate platforms, web and mobile. Web events are logged directly into Apache Kafka, while mobile events are routed through Google Cloud Pub/Sub. Once consumed by the pipeline, individual team members' score events are grouped by team (red or blue), windowed into hourly tumbling/fixed windows, summed for each platform individually (web or mobile) as well as globally across both platforms, then joined together into a single hourly summary per team. We'll discuss more details of the pipeline inline throughout the rest of this section.

4.1 System Model

We model a stream processing system as a dataflow graph, with a set of nodes and directed edges connecting them. Nodes have access to local state, receive messages on their incoming edges, and send messages on their outgoing edges. We assume that all activity in the dataflow graph occurs within a totally-ordered time domain, and message order is preserved on all edges. Finally, nodes can interact with external systems as part of their processing, e.g. by sending and receiving messages over a network.

Programmers of stream processing systems often rely on a framework to abstract away much of the complexity involved in making

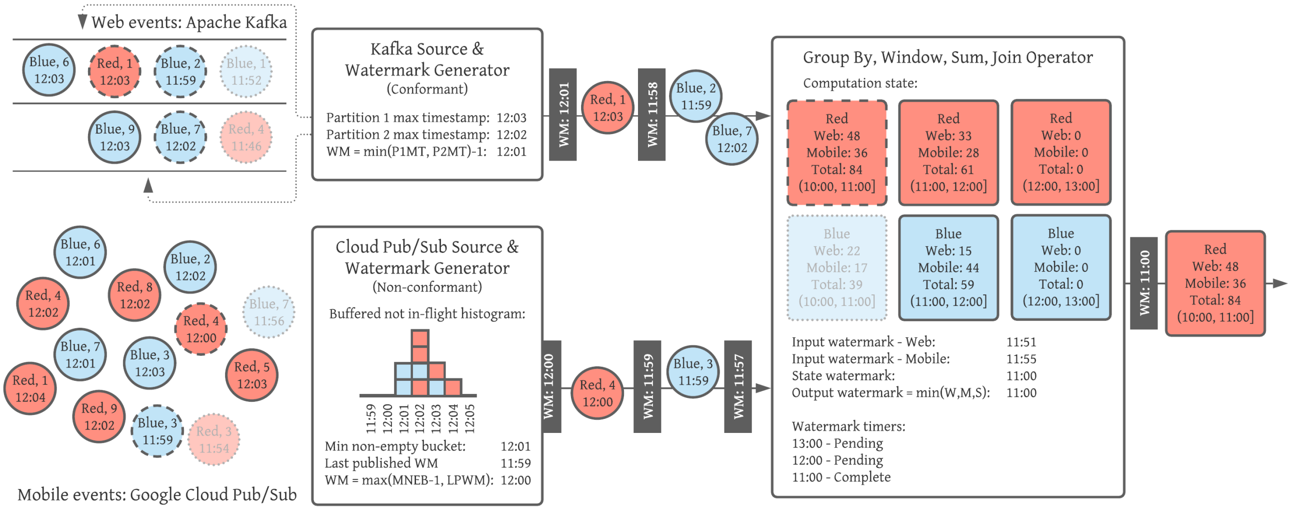

Web events: Apache Kafka
Blue, 6 12:03
Red, 1 12:03
Blue, 2 11:59
Blue, 1 11:52
Blue, 9 12:03
Blue, 7 12:02
Red, 4 11:46
Kafka Source & Watermark Generator
(Conformant)
Partition 1 max timestamp: 12:03
Partition 2 max timestamp: 12:02
WM = min(P1MT, P2MT)-1: 12:01
WM: 12:01
Red, 1 12:03
WM: 11:58
Blue, 2 11:59
Blue, 7 12:02
Blue, 6 12:01
Blue, 2 12:02
Red, 4 12:02
Red, 8 12:02
Red, 4 12:00
Blue, 7 11:56
Blue, 7 12:01
Blue, 3 12:03
Red, 5 12:03
Red, 1 12:04
Red, 9 12:02
Blue, 3 11:59
Red, 3 11:54
Mobile events: Google Cloud Pub/Sub
Cloud Pub/Sub Source & Watermark Generator
(Non-conformant)
Buffered not in-flight histogram:
11:59
12:00
12:01
12:02
12:03
12:04
12:05
Min non-empty bucket: 12:01
Last published WM 11:59
WM = max(MNEB-1, LPWM): 12:00
WM: 12:00
Red, 4 12:00
WM: 11:59
Blue, 3 11:59
WM: 11:57
Group By, Window, Sum, Join Operator
Computation state:
Red Web: 48 Mobile: 36 Total: 84 (10:00, 11:00]
Red Web: 33 Mobile: 28 Total: 61 (11:00, 12:00]
Red Web: 0 Mobile: 0 Total: 0 (12:00, 13:00]
Blue Web: 22 Mobile: 17 Total: 39 (10:00, 11:00]
Blue Web: 15 Mobile: 44 Total: 59 (11:00, 12:00]
Blue Web: 0 Mobile: 0 Total: 0 (12:00, 13:00]
Input watermark - Web: 11:51
Input watermark - Mobile: 11:55
State watermark: 11:00
Output watermark = min(W,M,S): 11:00
Watermark timers:
13:00 - Pending
12:00 - Pending
11:00 - Complete
WM: 11:00
Red Web: 48 Mobile: 36 Total: 84 (10:00, 11:00]

Figure 1: Example streaming analytics pipeline at time t_0, computing hourly per-team score aggregates. Individual score events for each team member are represented as circles containing the team name, score, and event time. Watermark updates are black rectangles. Summary aggregates are rounded rectangles. Sources, watermark generators, and operators are labeled as such. Faded events with dotted outlines have been processed already by the corresponding operator, and events with dashed outlines are in-flight; these markings help illustrate progress over time when comparing against Figure 2.

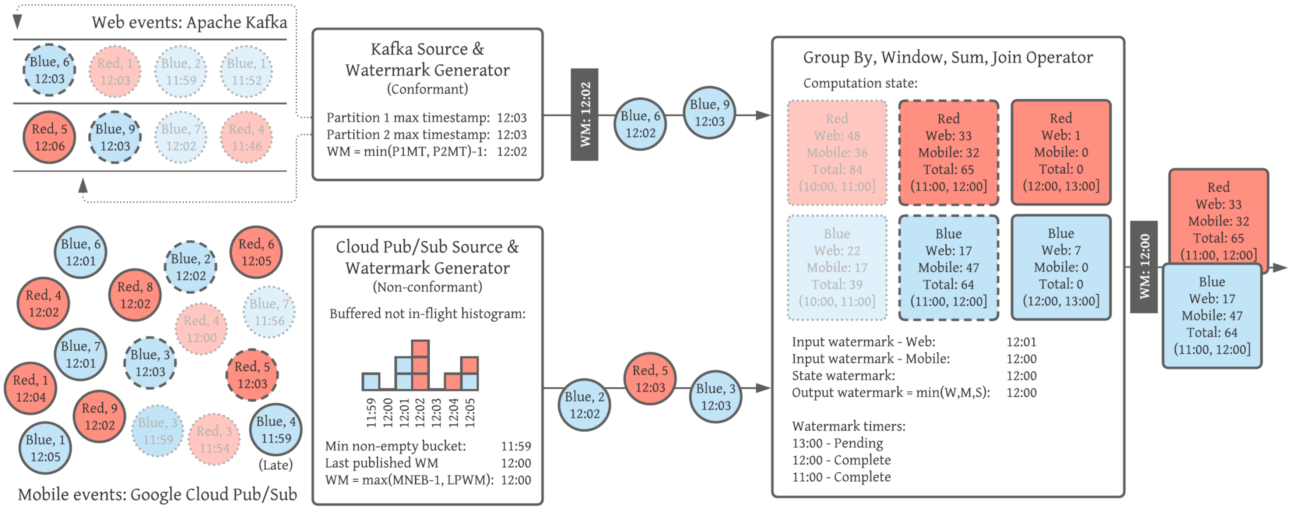

Web events: Apache Kafka
Blue, 6 12:03
Red, 1 12:03
Blue, 2 11:59
Blue, 1 11:52
Red, 5 12:06
Blue, 9 12:03
Blue, 7 12:02
Red, 4 11:46
Kafka Source & Watermark Generator
(Conformant)
Partition 1 max timestamp: 12:03
Partition 2 max timestamp: 12:03
WM = min(P1MT, P2MT)-1: 12:02
WM: 12:02
Blue, 6 12:02
Blue, 9 12:03
Blue, 6 12:01
Blue, 2 12:02
Red, 6 12:05
Red, 4 12:02
Red, 8 12:02
Red, 4 12:00
Blue, 7 11:56
Blue, 7 12:01
Blue, 3 12:03
Red, 5 12:03
Red, 1 12:04
Red, 9 12:02
Blue, 3 11:59
Red, 3 11:54
Blue, 4 11:59
(Late)
Blue, 1 12:05
Mobile events: Google Cloud Pub/Sub
Cloud Pub/Sub Source & Watermark Generator
(Non-conformant)
Buffered not in-flight histogram:
11:59
12:00
12:01
12:02
12:03
12:04
12:05
Min non-empty bucket: 11:59
Last published WM 12:00
WM = max(MNEB-1, LPWM): 12:00
Blue, 2 12:02
Red, 5 12:03
Blue, 3 12:03
Group By, Window, Sum, Join Operator
Computation state:
Red Web: 48 Mobile: 36 Total: 84 (10:00, 11:00]
Red Web: 33 Mobile: 32 Total: 65 (11:00, 12:00]
Red Web: 1 Mobile: 0 Total: 0 (12:00, 13:00]
Blue Web: 22 Mobile: 17 Total: 39 (10:00, 11:00]
Blue Web: 17 Mobile: 47 Total: 64 (11:00, 12:00]
Blue Web: 7 Mobile: 0 Total: 0 (12:00, 13:00]
Input watermark - Web: 12:01
Input watermark - Mobile: 12:00
State watermark: 12:00
Output watermark = min(W,M,S): 12:00
Watermark timers:
13:00 - Pending
12:00 - Complete
11:00 - Complete
WM: 12:00
Red Web: 33 Mobile: 32 Total: 65 (11:00, 12:00]
Blue Web: 17 Mobile: 47 Total: 64 (11:00, 12:00]

Figure 2: Example pipeline at time $t_1 > t_0$. In this figure, time has advanced such that the input watermark has reached 12:00 at the aggregation operator. As a result, the hourly summaries for the (11:00, 12:00] windows have been emitted downstream, and the output watermark advanced to 12:00 as well. Sources and watermark generators have progressed, with Kafka offsets and watermarks increased per partition, and the histogram of pending Pub/Sub timestamps evolved. Some new data has arrived. Note the late (Blue, 4, 11:59) event in the mobile source, rendering that watermark non-conformant.

such systems scalable and fault-tolerant. Programmers specify their program in terms of the framework's high-level APIs. The framework compiles and deploys this specification into a corresponding dataflow graph. In this section, we avoid the specifics of any individual framework for generality; Section 5 describes two particular implementations.

4.2 Interpreting Watermarks

Based only upon its formal definition, the idea of a watermark can seem arbitrary and uninteresting. However, from the intended use cases, one can infer the contexts in which useful watermarks arise. As previously noted, a watermark is a measure of a dataset's completeness with respect to time, over time. For that idea to make sense, we must be able to interpret the dataset as follows.

(1) The elements of the dataset represent events that occurred in the real world.
(2) The time of those events was measured using one of a set of adequately synchronized clocks.
(3) That measured time is inscribed as each element's *event time*.
(4) Each element is introduced into our stream processing system as a message, a process that can delay or reorder the elements relative to their event time.

Inside the streaming system, one can think of our abstract dataset as the set of all messages that ever were or will be sent on an edge. One can interpret the watermark on that edge as information about which real-world events have not yet been seen by the receiving node. This is exemplified in Figures 1 and 2, where watermark updates are only produced once all events known to the source with event times less than or equal to the watermark are in-flight. So interpreted, the watermark can be applied as discussed in Section 2.1.

4.3 How Watermarks are Generated

When a node of a dataflow graph introduces elements to the streaming system from an external event source, it must generate a corresponding watermark. To do so, one must take account of the elements (events) of the dataset and track their introduction into the streaming system. If an element is delayed, the watermark is too. If an element is lost (assuming this is tolerable), it must be excluded from consideration in the watermark. Whenever all elements with event time less than some time w have been sent as messages, the watermark for that node can advance to w.

In practice, generating a watermark that satisfies the conformance property (a "conformant" watermark) is challenging. An event dataset often traverses numerous systems before finally reaching the streaming system, such as the devices on which the events occur, logging systems, storage systems, and message buses. Support for this propagation in current systems is limited, so ensuring that all elements are properly accounted and the information is propagated through these systems is a challenging engineering problem.

In Figure 1, web events are logged directly to Kafka by a static set of frontends, each of them writing events with timestamps in monotonically increasing order, effectively ensuring that all necessary ordering information is retained. In this configuration, it's possible to compute a conformant watermark as the `min` of the largest timestamp thus far encountered in each partition, minus one, to account for the possibility of the next event having an equal timestamp to the current max.

Another case where it is feasible to generate conformant watermarks is within the streaming system itself. Some streaming systems measure the time of events happening within the system, such as messages being sent or received by the system, state being read or written, etc. They piggyback those measurements atop other messages, tracking separate watermarks for system-event times and application-event times. This results in "system watermarks", which are useful for certain types of garbage collection and health monitoring, as discussed in Section 2.1.

In cases where guaranteeing conformance is impracticable, programmers must rely on heuristics to predict delays and losses in a dataset to generate watermarks from within the streaming system. When using a non-conformant watermark, the system may introduce elements older than the watermark. We refer to these elements as **late data**. Generally, late data is undesirable, so users try to balance latency requirements against quantity of late data.

A common watermark heuristic is to assume **bounded disorder** in the process of transporting events from sources to the system. Under this assumption, when an element with an event time t is introduced to a node, the node advances its watermark to $t - \Delta$, where Δ is a configurable constant. Another common heuristic is the **timeout**, where the watermark is advanced to t after waiting for some constant duration to elapse after the first element with event time t is introduced. We have found both of these heuristics to be problematic in practice, as they introduce unnecessary delays when the system is running well and not enough delay when problems arise, yielding large amounts of late data.

A more sophisticated heuristic is to **statistically model** the behavior of event sources, and adapt the watermark delay accordingly. The node calculates a rolling histogram of element lag (similar to the histograms in Figures 1 and 2), which is the difference between the arrival time and event time of an element. This histogram approximates the distribution of the delay between an event and its introduction to the system. The node fits a statistical model (e.g. a Gamma distribution) to the histogram and delays the watermark by the duration corresponding to a quantile chosen according to application requirements (e.g. 0.999). By re-evaluating the fit periodically, the watermark can adapt to changes in the behavior of the source, like delays caused by congestion or outages. This kind of approach takes considerably more effort than the simple heuristics above, but we have found it to substantially improve watermark latency while yielding less late data.

In Figures 1 and 2, the watermark generator for mobile events constructs a histogram tracking all known event times for data buffered in Pub/Sub and not in-flight downstream. The generator then uses the timestamp of the smallest non-empty bucket in the histogram as the basis for its watermark. However, because mobile events may be arbitrarily delayed, it's possible for data to arrive late. We see this in Figure 2, where an event with timestamp 11:59 arrives in Pub/Sub after the watermark generator advances the watermark to 12:00. The source must latch the watermark to 12:00 to maintain monotonicity, violating conformance. Strategies for dealing with late data due to non-conformant watermarks are discussed in Section 4.5.

4.4 How Watermarks Propagate

Assuming all nodes that introduce new elements are able to generate adequate watermarks, we turn to the question of how to compute the watermarks of other nodes in the graph. The goal for these derived watermarks is to advance them quickly without making any more data late. Here, we discuss the constraints implied by this goal on the derived watermarks, deferring a discussion of specific algorithms to Section 5.

It is useful to generalize the definition of watermarks (given in Section 2) to include *edges* in the dataflow graph. The definition is similar to that for a node but restricts the elements considered to those sent along the edge.

From their definitions, we can conclude that the watermark for a node must not be less than that for any of its incoming edges. The watermark for an edge depends on the messages that its source node will send in the future, which, of course, depends on the logic within that node. For arbitrary node logic, that's as much as we can conclude, but there are several common patterns that are worth exploring.

First, consider nodes that perform stateless, element-wise computation, sending messages with unchanged timestamps. The watermark for an edge outgoing from such a node is only constrained to be no greater than that of its source node. If the watermark of the source advances, that means that it will receive no messages with an earlier timestamp, which means the node will not send any such messages either.

Second, consider nodes that aggregate messages using a node's local state (e.g. a some type of a node-local store, serving as a temporary processing memory). This pattern is common when implementing `JOIN`, `GROUP BY`, or `PARTITION BY` operators. As messages arrive at a node, it stores them or a partial aggregate in its state until the watermark indicates that all messages in an aggregate have arrived, at which point the final aggregate is sent. This final aggregate usually has an event time that is a function of the event times of the messages that comprise it, like `max` or `min`. Here, the watermark must be held back by applying the timestamp aggregating function to the stored data (either buffered messages or partial aggregates) and the node's watermark. For example, if the function is `max`, the watermark for outgoing edges must not exceed $\mathtt{max}(\{w\} \cup S)$, where w is the node's watermark and S is the set of event times in the node's state. We see this in Figures 1 and 2, where the aggregation operator holds the watermark to the `min` of its input watermarks and the start times for all incomplete hourly windows buffered in state.

In general, frameworks cannot automatically infer watermark propagation constraints. The use of pattern-specific methods is preferred when possible, as direct manipulation of the watermark is error prone.

4.5 How Watermarks are Consumed

At this point, we have explained how to compute watermarks for every node and edge in a dataflow graph. The remaining question is "How should they be used?"

The most general answer is that nodes can read the watermark and incorporate its value into their processing. A typical interface for this functionality is to allow programmers to schedule callbacks when a node's watermark reaches or exceeds a certain value. We call such callbacks **watermark timers** by analogy with traditional, real-time timers. For example, if a node is computing the number of messages with event time between t_0 and t_f, it can store a running count of the number of such messages seen, and register a watermark timer for t_f. When the watermark reaches or exceeds t_f, the callback sends the current value of the counter on an outbound edge, which is guaranteed to include all messages within the interval.

This is another pattern that can be captured by higher-level APIs: if a program is aggregating a set of elements whose timestamps have a known upper bound, a framework can automatically schedule watermark timers for each aggregate and send the final aggregate. Such aggregates are known as "windows" in Apache Beam and Apache Flink, overloading the term as used in SQL and complex event processing. We see this in Figures 1 and 2, where watermark timers are set for the end of each hourly window. Upon triggering, the corresponding aggregates are emitted downstream.

Non-conformant watermarks complicate processing that depends on watermarks. They weaken the guarantee of completeness, turning it into an educated guess from the data source, and leading to the production of late data. Despite the added complexity in downstream code, the use of non-conformant watermarks can be a worthwhile trade-off. A number of techniques are known to deal with late data which are typically easier than instrumenting data sources with completeness information.

The most basic technique is to simply **ignore late data** and compute an approximate result. This technique is useful because it makes a non-conformant watermark conformant, which relieves downstream nodes from having to take late data into account. But there are substantial downsides. The results of such programs are not guaranteed to be correct, and likely depend on the timing of input elements. Use cases that do not require highly accurate results can often tolerate these disadvantages. However, it is prudent to vet the heuristics used to compute the watermarks of a pipeline's sources and instrument the pipeline to monitor how much late data is dropped; the next best thing to having accurate results is knowing their inaccuracy.

Another technique is to **recompute results** when late data change them. This creates two challenges: increased resource consumption and complexity. The increased resource consumption comes from retaining results while waiting for late data to arrive and the additional processing when they do. The complexity comes from converting program logic into an incremental form that can react to late data correctly and efficiently. Some frameworks, such as Differential Dataflow, can do this conversion automatically, but most, including Beam, Kafka Streams, Flink, and Spark rely on programmers to ensure the program is correctly incrementalized. Once these challenges are overcome, the resulting program is robust to late data while still benefiting from the heuristic completeness provided by non-conformant watermarks.

5 IMPLEMENTATIONS

For the rest of the paper, we turn our focus to the watermark implementation in two modern stream processing systems, Apache Flink and Google Cloud Dataflow. These two systems are particularly interesting because they support essentially identical, sophisticated watermark semantics, but do so via markedly different approaches. In Flink, watermarks are computed and propagated through the pipeline along the data path, whereas in Cloud Dataflow watermarks are computed and propagated out of band via an external aggregator.

Additionally, both systems are well supported by Apache Beam, a framework and protocol for defining data processing pipelines in several programming languages,[2] for execution on one of several

[2] Currently Java, Python, SQL, and Go.

data processing engines.[3] This affords us the opportunity to evaluate the execution of an identical Apache Beam pipeline using both Flink and Cloud Dataflow.

In this section, we compare the watermark implementations in Beam, Flink, and Cloud Dataflow from an architectural perspective. Then in Section 6, we compare Flink and Cloud Dataflow empirically by running a suite of Apache Beam benchmarks on both.

5.1 Watermarks in Beam

The main usage of Apache Beam in this paper is to provide a common mechanism for defining a pipeline for evaluation on Flink and Cloud Dataflow. But there are some interesting aspects of Beam's watermark implementation that are worth calling out before diving into a comparison of those two systems.

Firstly, to provide portability across engines, Beam includes its own generalized definition and protocol for managing watermarks and their affect on aggregation steps. To execute user-defined steps in a Beam pipeline, a data processing engine sends RPCs to a Beam *SDK harness*, which is a UDF server that performs computations and emits watermark-related metadata.

Secondly, a unique feature of Beam is the so-called *Splittable DoFn*: data sources which are not roots of the data processing graph, but intermediate nodes. For example, a user-defined data processing step may accept a stream of URLs indicating a dynamic set of event streams to be read and produced downstream. The Beam protocols allow individual input elements to result in unbounded outputs, while maintaining checkpoints and updating watermarks. Thus, the Beam UDF server is intimately involved in watermark processing.

5.2 Watermarks in Flink & Cloud Dataflow

Apache Flink and Google Cloud Dataflow are distributed data stream processors that both implement a watermark mechanism supporting event-time processing semantics and have much in common architecturally. Flink and Cloud Dataflow programs are defined as directed, acyclic graphs. Graph nodes represent stateful data processing operators and directed edges represent the data channels between operators. Each operator in the dataflow graph is translated in one or more execution nodes, which are distributed throughout a cluster of servers. Node state and input data are partitioned across these nodes to be processed in parallel. The execution nodes send and receive messages via communication edges.

Despite this similarity, the two systems diverge significantly in their watermark implementations: Flink represents watermarks as metadata inside the data streams themselves, while Cloud Dataflow computes and propagates watermarks out-of-band via a separate aggregator node. We elaborate on the two approaches by looking at how they generate, propagate, and consume watermarks.

5.2.1 Generating Watermarks. **Flink**: A Flink program can generate watermarks in its source nodes or in dedicated watermark generation nodes. Source nodes may compute watermarks based on the ingested elements or leverage metadata provided by the system from which they read their input elements, such as system-managed partitions, offsets, or timestamps. Dedicated watermark generation nodes can compute watermarks only based on the timestamps of the input elements they have observed.

Flink features two mechanisms to publish watermarks. Nodes can continuously maintain a watermark and publish it periodically based on a configurable watermark interval. Alternatively, nodes may directly publish watermarks, possibly for every input record they process. Flink guarantees the strict monotonicity of watermarks by only forwarding watermarks that are larger than the previous watermark.

Cloud Dataflow: Cloud Dataflow relies on Beam's source frameworks, namely `UnboundedSource` and `SplittableDoFn`, to generate watermarks. `UnboundedSource` provides the means for establishing watermarks at root nodes of the pipeline graph, whereas `SplittableDoFn` affords the ability to establish (or re-establish) watermarks at any point in the pipeline. The Beam protocol for setting the watermark uses *watermark holds*, timestamps stored in a node's state that prevent its watermark from advancing.

Cloud Dataflow utilizes a single mechanism for publishing watermarks, more akin to the first approach in Flink: nodes continuously maintain a local watermark and publish it periodically based on a system-wide watermark interval to a global watermark aggregator. As in Flink, Cloud Dataflow ensures strict montonicity of watermarks.

5.2.2 Propagating Watermarks. **Flink**: Flink nodes propagate watermarks by emitting them as special metadata messages alongside regular data output. Since Flink's communication edges preserve the order of sent messages, a receiver node ingests watermarks and data messages in the same order in which they were emitted by the sender node.

Each node keeps track of the maximum watermark received on each of its input edges. The watermark of a node is computed by taking its smallest input-edge watermark. When a node receives a new watermark message, it updates the corresponding input-edge watermark and checks whether it needs to update its node watermark.

Flink's processing nodes do not persist watermark metadata. When a node is restarted to recover from a failure, its watermark is set to a low-valued constant. The watermark is set once the node receives a new watermark message on all of its input edges.

Cloud Dataflow: Cloud Dataflow is dynamically load balanced. Instead of static partitioning, each operator's data stream is dynamically range partitioned, producing an evolving data flow graph where nodes can split, merge, and move between workers. Each node keeps track of the current sets of unprocessed records and watermark holds. These are summarized into histograms, which are both kept in memory and stored persistently. The watermark for a node is computed as the minimum of the lower bound of these histograms and the watermarks of input edges.

Each node periodically reports the current watermark to a central system that aggregates the watermark reporting over all nodes. This central system is partitioned by one or more operators, and so can handle a substantial reporting load. The watermark for an operator is computed as the minimum over all of its node watermarks. This aggregated operator watermark is sent to nodes with input edges from that operator, where it is used as the input-edge watermark.

[3] Currently there exist production runners for Apache Spark, Apache Flink, Apache Samza, Apache Nemo, Hazelcast Jet, Twister2, Google Cloud Dataflow, and IBM Streams. Draft runners exist for Apache Tez, Apache Hadoop MapReduce, and JStorm

5.2.3 Reacting to Advancing Watermarks. Flink and Cloud Dataflow both reduce all watermark-based processing into executing watermark timer callbacks. A node sets timers according to its application requirements, either explicitly via watermark timer APIs, or implicitly via windowing and watermark trigger APIs. When a node's watermark advances, it processes all unprocessed timers with timestamps less than the watermark's new value, invoking the corresponding callback for each timer. These callbacks may send messages or modify state. This process is essentially identical in both systems. Once timers have been processed, the new watermark value is propagated to all successive nodes.

5.2.4 Design Tradeoffs. The two differing approaches to watermark implementations in Flink and Cloud Dataflow result in some interesting design tradeoffs. We discuss a few of these here, as well as some challenges to which both approaches are susceptible.

Watermark reporting latency is slightly higher for Cloud Dataflow due to the extra network hop required for watermark updates to reach the central watermark aggregator, as well as the time spent on persisting watermark progress to persistent state. This contrasts with Flink's approach of embedding watermark updates within data streams and never persisting them.

Fault recovery is slower in Flink than in Cloud Dataflow due to the finer granularity at which Cloud Dataflow's state (including watermark progress) is checkpointed. When a single Cloud Dataflow node fails, a replacement node must be brought online, after which it can read its state to initialize watermark values. In contrast, when a single Flink node fails, a replacement node must be brought online, and the entire pipeline must halt, rewind, and resume from the last completed checkpoint before the failure occurred. Watermarks are reset to the beginning of time and must be propagated once more from sources before processing can resume.

These design choices were the direct result of environmental assumptions made about where each system would typically be run: Cloud Dataflow's processing core runs on Google's Borg [21] cluster manager, where priority-driven preemption of running tasks is the norm, whereas Flink jobs were designed for execution on bare metal hardware or public cloud VMs (or a container manager running thereon), where preemption is non-existent or rare.

Consistency semantics are more complex to ensure in Cloud Dataflow due to the dynamic range partitioning and coordination with the external watermark aggregator. Cloud Dataflow relies upon a distributed ownership protocol to ensure only one worker may report watermark updates for a given processing key range at any one time. In contrast, Flink piggybacks on its existing work ownership protocols for data streams. And because Flink nodes ingest data messages and watermark messages from the same queue, no additional synchronization is required to prevent concurrent processing of data elements and timers.

Idle workers require special handling in Flink. A source node without watermark progress can affect the progress of the whole program. To mitigate the problem of a source task without input data, Flink source nodes can declare themselves as idle, which means that their output channels are temporarily excluded when subsequent nodes update their watermark. Source nodes become active again as soon as they continue to emit elements. Cloud Dataflow watermark updates continue even in the absence of data, and a source node that currently has no data to deliver will publish a watermark which continues to track current system time.

Bottlenecks are problematic for both systems. The speed at which the watermark of a node advances depends on the watermark update speed of its slowest predecessor. This means that, depending on the program, a single overloaded or bottle-necked node can obstruct the progress of all downstream nodes in the graph.

Watermark skew is another common problem. For example, two source nodes can read different partitions of the same source or from completely unrelated sources. In either case, watermark progress is limited to the source node with the lowest watermark and slowest progress. Unaligned source watermarks can lead to a significant increase in state size to buffer in-flight data. Flink and Cloud Dataflow both utilize skew-control synchronization mechanisms across nodes that can be used to coordinate data ingestion such that faster source nodes hold data back in the source system to keep their watermarks more closely aligned.

6 EVALUATION

In this section, we evaluate the results of running a single Apache Beam pipeline on both Flink and Cloud Dataflow using a number of different configurations. Our experiments are meant to verify that the implementations of watermarks scale reasonably as the underlying data processing workload scales. What are the implications on latency as a function of scale? There are many possible dimensions of scale, but the most important in practice are (a) the **number of workers** in a job, (b) the amount of **throughput** handled by the job, and (c) the **depth** of the job's topology. Our experimental configurations are chosen to hold one or more of the dimensions constant while varying the others. The details of the pipeline logic don't affect watermark propagation; only the overall load on the system has an effect.

Our experimental setup consists of a Beam pipeline[4] that:

(1) Generates and assigns message timestamps at the beginning of the pipeline at a constant rate. The messages are timestamped to wall-clock time - this is done to avoid extraneous delays associated with sources from affecting our measurements.
(2) Shuffles messages uniformly across a set of 1000 distinct keys.
(3) Windows the messages into 1-second windows after the shuffle, and delivers individual window results when watermark advancement indicates that a given window has closed. Note that since the messages are generated internally to the pipeline, this is a conformant watermark, and there will be no late messages.
(4) Repeats the shuffle-and-window steps another two times.
(5) Measures the difference between each window's maximum timestamp and the time it was produced. This difference is the latency induced by the watermark system when propagating watermarks from one window task across a shuffle stage to the next window task.

The pipeline was implemented with Apache Beam 2.27.0 and executed with Beam's runners for Flink and Cloud Dataflow. Experiments were run in three configurations:

[4] Code available at: http://s.apache.org/watermarks-paper-beam-pipeline

- Number of workers varied.
- Throughput varied.
- Pipeline depth varied.

Results[5] measure the median and 95th-percentile latencies of window materialization caused by watermark advancement at different stages of the job.

We ran our Flink experiments with Apache Flink 1.12.1 on Google Compute Engine. Each worker was configured with a single processing slot and ran on an n1-standard-2 node with 2 vCPUs and 7.5GB RAM. We enabled unaligned checkpoints and configured a checkpointing interval of ten seconds, i.e., every ten seconds a full copy of the pipeline's state and in-flight data was taken.

We ran our Cloud Dataflow experiments in early February with Apache Beam 2.27. Each user worker also ran on an n1-standard-2 node using Cloud Dataflow's Streaming Engine service.

In these experiments, we are only measuring watermark propagation latency; end to end latency of data being available downstream depends on a number of factors including how disordered the input streams are and whether the sinks require waiting for checkpoints to produce output.

6.1 Watermark Latency vs. Worker Count

In our first configuration, we measured observed changes to watermark latency as worker count was varied from four to 128 workers, with throughput held constant at 1000 messages per second, 24 bytes per message, and pipeline depth of three shuffles.

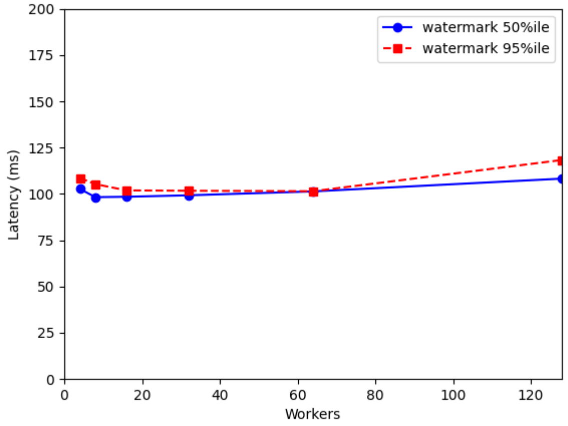
Figure 3 shows the results for the Flink and Cloud Dataflow versions of the pipeline, respectively, with median and 95th-percentile latencies captured following each of the three shuffle stages.

Flink: In Flink, each shuffle stage adds approximately 100 ms median latency. The increasing number of workers has only a marginal effect on median and 95th-percentile latencies, but the numbers do show an upwards trend as worker count increases.

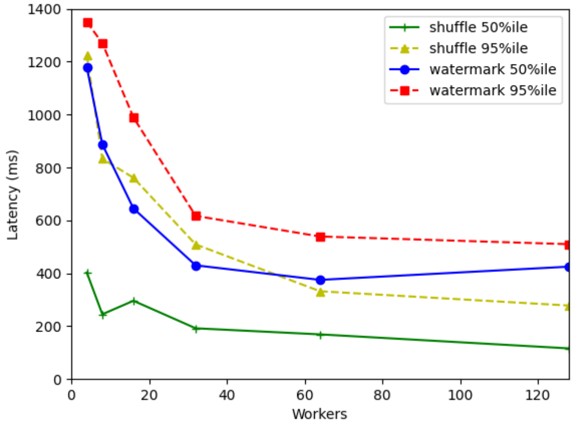
Dataflow: In Cloud Dataflow, each shuffle stage adds approximately 500-1000 ms median latency. Conversely to Flink, increasing the number of workers *decreases* the latency of window materialization. This is expected since as we increase the number of workers while holding the throughput constant, we increase available parallelism, spreading out the work. This also demonstrates that with the currently-tested scale, we do not enter into a region where the number of workers starts to noticeably affect the additional delay due to watermark aggregation.

6.1.1 Cloud Dataflow Constants and Shuffle Latency. As a brief aside, we discovered two interesting corollary results with Cloud Dataflow as part of these experiments.

Firstly, initial experiments showed no variation at all with scale. However we quickly found that this was the side-effect of default turnings in Cloud Dataflow's watermark aggregation batching being set too conservatively. Cloud Dataflow has tunings around the periodicity of aggregation and bucketization of timestamps that err on the side of higher latency to avoid potential scalability drawbacks. With the default tunings, all configurations resulted in a latency around 3 seconds per shuffle+window, which was effectively measuring the sum of the periodicity and bucketization tuning parameters. Adjusting all of the relevant tunings down to 10ms values yielded better results that demonstrated the effect of scale on latency. Similar parameter tunings will be pushed to production in the future.



Flink



Cloud Dataflow

Figure 3: Latency vs. Worker Count

Secondly, when we measure the latency of window materialization in Cloud Dataflow, we are also measuring the latency of the associated `GroupByKey` shuffle, so for additional insight, we measured and compared those numbers independently.[6]

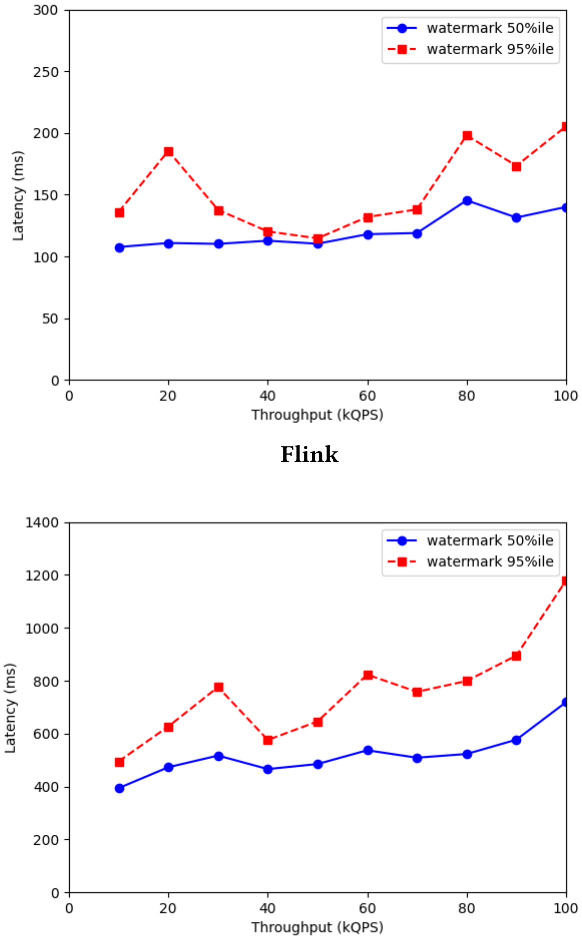
Here we note that the window materialization latency closely tracks the higher-percentiles of shuffle latency - being generally close to the 95th percentile of shuffle latency. This is expected, since all data for a window must be shuffled before the window can be materialized. This also shows that the latency of window materialization is dominated by the latency of the shuffle and the additional latency of aggregating and broadcasting watermarks is comparatively insignificant.

[5] Raw data available at: http://s.apache.org/watermarks-paper-evaluation-data

[6] Since Flink propagates watermarks inline with data, it is not possible to report separate latencies for data shuffle and watermark updates.

6.2 Watermark Latency vs Throughput

In our second configuration, we measured observed changes to watermark latency as throughput was varied from 10,000 to 100,000 messages per second at 24 bytes per message, with worker count held constant at 128 worker nodes and three shuffles.



Flink

Cloud Dataflow

Figure 4: Latency vs. Throughput

Figure 4 shows the results for the Flink and Dataflow versions of the pipeline, respectively, with the median and 95th-percentile latencies captured for shuffle stages between two consecutive window tasks. For Dataflow, watermark latency tracked closely with shuffle latency just as in the Watermark Latency vs Worker Count experiment. We have already accounted for this effect in the previous experiment so we omit the separate data on shuffle here.

Flink: In the Flink pipeline, the watermark latency grows only moderately from 105 ms to 140 ms when increasing the throughput from 10,000 to 100,000 messages per second.

Cloud Dataflow: In the Cloud Dataflow pipeline, we observe that the window materialization latency does increase in the experiment where we hold the number of workers constant and vary throughput. Here too though, the latency is dominated by shuffle latency, and the effect of higher throughput on watermark aggregation appears negligible.

6.3 Watermark Latency vs. Pipeline Depth

Finally, we measured the effect of increasing the number of shuffle stages in the pipelines. In Dataflow, we observed no significant change in latency based on the number of shuffles, but in Flink, there was a clear relationship where higher numbers of workers and shuffles resulted in higher latency, as shown in Figure 5. We verified that the effect is not caused by a saturated network, checkpointing overhead, or backpressured tasks and hypothesize that it might be due to the watermark processing subsystem being overloaded by the overall volume of watermark traffic.

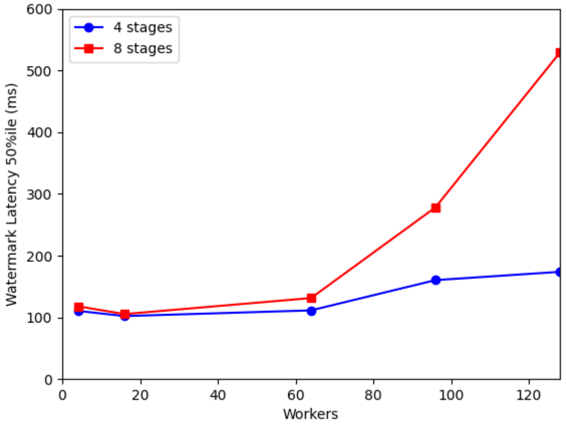


Figure 5: Flink Latency vs Pipeline Depth

6.4 Takeaways

Finally, an important contrast between the implementations is that, overall, the Dataflow latency numbers are somewhat higher. This is due to the different implementations of checkpointing. Cloud Dataflow checkpoints messages that are being shuffled "in-line," meaning that the latency here includes the latency of the writes associated with the checkpointing of the data. On the other hand, Flink globally checkpoints data periodically.

7 FUTURE WORK

One key part of incorporating watermarks into a system is the method of watermark generation for the data sources in a pipeline. Currently, in Apache Beam, this is done on an ad-hoc basis for each type of source. For example, when reading from an *ordered-by-partition* source like Kafka, we can assume ordering of timestamps per partition and use the minimum of the latest timestamp seen across partitions. However, this makes writing new sources difficult, as this logic needs to be designed and implemented separately for each new source. Future work could address this by finding common patterns across sources which can be generalized easily to new ones.

Another area of future work is to quantify the differences in fault-tolerance of the watermark systems between Flink and Cloud Dataflow. In Section 5.2.4, we discussed these differences qualitatively, but the precise costs are not completely known.

8 SUMMARY

Watermarks represent the temporal completeness of an out-of-order data stream. Reasoning about the completeness of infinite streams is one of most critical challenges faced by stream processing systems. It's also one of the least understood and least adequately addressed. Compared to other approaches for dealing with completeness of unbounded data streams (order-agnostic processing, ordered processing, timestamp frontiers, punctuations), we believe watermarks provide the best balance of cost/benefit tradeoffs.

Watermarks allow the system to support important streaming use cases that require providing a single authoritative answer (such as notifications), require reasoning about a *lack* of data (e.g., dip detection), or require expensive or non-incremental processing (e.g., `MAX` of a `SUM` of signed integers). They provide important signals that allow a streaming system to efficiently and safely garbage collect unneeded inputs and state. And they act as a remarkably general first-alert system for issues occurring within data streaming pipelines, often highlighting when and where a problem is occurring, regardless of the underlying root cause.

Watermarks afford all of this in the face of dynamically changing input disorder, and while still allowing incremental processing to proceed as the system awaits a completeness signal. They may be applied very generally to a broad class of input source types using heuristic or statistical algorithms, or fine-tuned with specialized watermark generation logic tailored to the unique characteristics of a given source.

Watermarks do have their shortcomings. A common pain point is watermark generation algorithms which are overly conservative, resulting in unwanted delays. But such issues can often be addressed either by fine-tuning the watermark implementation, or relaxing the completeness constraints enforced by the watermark generator or system. Another common criticism of watermarks is their complexity, but we believe this is a false argument, as systems which opt out of tackling the challenges of completeness themselves simply push that complexity onto their users; far better for us as system builders to solve the problems of completeness within the framework itself, and relieve our users of that burden.

Furthermore, we believe that when properly understood, watermarks are not so scary and complex. Simply put, a watermark's current value informs a processor that all messages with a lower timestamp have been received. From an architectural perspective, there are just three core pieces of a functional watermark implementation: generation, propagation, and consumption.

Watermark generation is by far the most challenging of the three, due to the relatively open-ended nature of computing a completeness signal for unbounded streams of data. Approaches vary from simple heuristics (e.g., "max event time ever seen plus timeout delay" high watermarks available in Kafka Streams, Spark Structured Streaming, and one of Flink's watermark generator implementations), to more sophisticated algorithms (e.g., Adaptive Watermarks for Flink [5]), to custom logic hand-tuned to a specific input source's unique characteristics (e.g., a conformant watermark generator for a Kafka topic whose static set of partitions are known to contain monotonically increasing timestamps, or Cloud Dataflow's bespoke watermark generator for Google Pub/Sub [3]).

Once generated, watermark propagation is a classical distributed systems exercise: aggregate and deliver watermark updates through the pipeline in an efficient, scalable, and reliable manner. Interestingly enough, Flink and Cloud Dataflow provide two distinct approaches to this endeavor: one in-band with the data streams themselves, another out-of-band via an external aggregator.

Of course, a watermark signal is of no use unless it's acted upon. Of the three architectural pieces, watermark consumption is the most straightforward. In most cases, it involves invoking an action after a watermark advances beyond a target timestamp.

Taken together, these three pieces are all a system needs in order to provide a robust and flexible mechanism for reasoning about completeness. And as our experimental evaluation shows, the differing approaches taken by by Flink and Cloud Dataflow both yield respectable watermark latencies in the 100s of milliseconds range, even in the face of scaling worker counts and throughput.

There are differences between the two approaches. Cloud Dataflow tends to have slightly higher latencies (due to the extra network hop and persistent storage writes for checkpointing watermark progress); Flink latency appears to grow super-linearly as both pipeline depth and worker count increase (likely due to the increased volume of watermark traffic and a hidden inefficiency in the watermark subsystem). But the overall shape of the user experience provided by the two systems is remarkably similar, which again hints at the idea that sophisticated watermark systems are in fact quite tractable to build.

For Flink and Cloud Dataflow, watermarks have proven to be an indispensable tool that allows our users to tame some of the trickiest problems in stream processing. It is our hope that this paper helps shed some light and understanding on this misunderstood beast, and we look forward to seeing how the broader stream processing community continues to evolve watermarks over the coming years.

ACKNOWLEDGMENTS

We would like to thank Sam McVeety and Mehran Nazir for their time and thoughtful reviews of this work, as well as everyone who has contributed to Apache Beam, Apache Flink, Google Cloud Dataflow, and MillWheel for making this work possible.

We would also like to acknowledge the *Google for Education* program for supporting parts of this research with their grant of Google Cloud resources to Dr. Begoli. We would also like to thank Apache Software Foundation (ASF) for their support.

This manuscript has been in part co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

REFERENCES

[1] 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. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[2] 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. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[3] T. Akidau, S. Chernyak, and R. Lax. *Streaming Systems*. O'Reilly Media, Inc., 1st edition, 2018.

[4] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. Etalis: Rule-based reasoning in event processing. In *Reasoning in event-based distributed systems*, pages 99–124. Springer, 2011.

[5] A. Awad, J. Traub, and S. Sakr. Adaptive watermarks: A concept drift-based approach for predicting event-time progress in data streams. In *EDBT*, pages 622–625, 2019.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[7] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[8] T. Das. Event-time aggregation and watermarking in apache spark's structured streaming. https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apacmhe-sparks-structured-streaming.html, 2017. [Online; accessed 06-Feb-2021].

[9] M. J. S. Eno Thereska, Michael Noll. Watermarks, tables, event time, and the dataflow model. https://www.confluent.io/blog/watermarks-tables-event-time-dataflow-model/, 2017. [Online; accessed 25-Jan-2021].

[10] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. Sase: Complex event processing over streams. *arXiv preprint cs/0612128*, 2006.

[11] C. S. Jensen and R. Snodgrass. Temporal specialization and generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–974, 1994.

[12] K. Kulkarni and J.-E. Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, Oct. 2012.

[13] R. Lax. After lambda: Exactly-once processing in cloud dataflow, part 2 (ensuring low latency). https://cloud.google.com/blog/products/gcp/after-lambda-exactly-once-processing-in-cloud-dataflow-part-2-ensuring-low-latency, 2017. [Online; accessed 06-Feb-2021].

[14] F. McSherry. Timelydataflow. https://github.com/TimelyDataflow/timely-dataflow, 2020.

[15] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[17] J. Roesler. Kafka streams' take on watermarks and triggers. https://www.confluent.io/blog/kafka-streams-take-on-watermarks-and-triggers/, 2019. [Online; accessed 25-Jan-2021].

[18] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274, 2004.

[19] J. Teich, L. Thiele, and E. A. Lee. Modeling and simulation of heterogeneous real-time systems based on a deterministic discrete event model. In *Proceedings of the 8th international symposium on System synthesis*, pages 156–161, 1995.

[20] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[21] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[22] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, 2021.

[23] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, 2006.