The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak,
Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills,
Frances Perry, Eric Schmidt, Sam Whittle
Google
{takidau, robertwb, chambers, chernyak, rfernand,
relax, sgmc, millsd, fjp, cloude, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of approach is necessary to deal with these evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have seen all of our data, only that new data will arrive, old data may be retracted, and the only way to make this problem tractable is via principled abstractions that allow the practitioner the choice of appropriate tradeoffs along the axes of interest: correctness, latency, and cost.

In this paper, we present one such approach, the Dataflow Model[1], along with a detailed examination of the semantics it enables, an overview of the core principles that guided its design, and a validation of the model itself via the real-world experiences that led to its development.

[1]We use the term “Dataflow Model” to describe the processing model of Google Cloud Dataflow [20], which is based upon technology from FlumeJava [12] and MillWheel [2].

Proceedings of the VLDB Endowment, Vol. 8, No. 12

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser each day, as well as aggregate statistics about the videos and ads. In addition, they want to efficiently run offline experiments over large swaths of historical data.

Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. They also want to know how much they are being charged/paid. They want all of this information as quickly as possible, so that they can adjust budgets and bids, change targeting, tweak campaigns, and plan future directions in as close to real time as possible. Since money is involved, correctness is paramount.

Though data processing systems are complex by nature, the video provider wants a programming model that is simple and flexible. And finally, since the Internet has so greatly expanded the reach of any business that can be parceled along its backbone, they also require a system that can handle the diaspora of global scale data.

The information that must be calculated for such a use case is essentially the time and length of each video viewing, who viewed it, and with which ad or content it was paired (i.e. per-user, per-video viewing *sessions*). Conceptually this is straightforward, yet existing models and systems all fall short of meeting the stated requirements.

Batch systems such as MapReduce (and its Hadoop variants, including Pig and Hive), FlumeJava, and Spark suffer from the latency problems inherent with collecting all input data into a batch before processing it. For many streaming systems, it is unclear how they would remain fault-tolerant at scale (Aurora [1], TelegraphCQ [14], Niagara [15], Esper [17]). Those that provide scalability and fault-tolerance fall short on expressiveness or correctness vectors. Many lack the ability to provide exactly-once semantics (Storm, Samza [7], Pulsar [26]), impacting correctness. Others simply lack the temporal primitives necessary for windowing[2] (Tigon [11]), or provide windowing semantics that are limited to tuple- or processing-time-based windows (Spark Streaming [34], Sonora [32], Trident [5]). Most that provide event-time-based windowing either rely on ordering (SQLStream [27]), or have limited window triggering[3] semantics in event-time mode (Stratosphere/Flink [3, 6]). CEDR [8] and Trill [13] are noteworthy in that they not only provide useful triggering semantics via punctuations [30, 28], but also provide an overall incremental model that is quite similar to the one we propose here; however, their windowing semantics are insufficient to express sessions, and their periodic punctuations are insufficient for some of the use cases in Section 3.3. MillWheel and Spark Streaming are both sufficiently scalable, fault-tolerant, and low-latency to act as reasonable substrates, but lack high-level programming models that make calculating event-time sessions straightforward. The only scalable system we are aware of that supports a high-level notion of unaligned windows[4] such as sessions is Pulsar, but that system fails to provide correctness, as noted above. Lambda Architecture [25] systems can achieve many of the desired requirements, but fail on the simplicity axis on account of having to build and maintain two systems. Summingbird [10] ameliorates this implementation complexity by abstracting the underlying batch and streaming systems behind a single interface, but in doing so imposes limitations on the types of computation that can be performed, and still requires double the operational complexity.

None of these shortcomings are intractable, and systems in active development will likely overcome them in due time. But we believe a major shortcoming of all the models and systems mentioned above (with exception given to CEDR and Trill), is that they focus on input data (unbounded or otherwise) as something which will at some point become complete. We believe this approach is fundamentally flawed when the realities of today's enormous, highly disordered datasets clash with the semantics and timeliness demanded by consumers. We also believe that any approach that is to have broad practical value across such a diverse and varied set of use cases as those that exist today (not to mention those lingering on the horizon) must provide simple, but powerful, tools for balancing the amount of correctness, latency, and cost appropriate for the specific use case at hand. Lastly, we believe it is time to move beyond the prevailing mindset of an execution engine dictating system semantics; properly designed and built batch, micro-batch, and streaming systems can all provide equal levels of correctness, and all three see widespread use in unbounded data processing today. Abstracted away beneath a model of sufficient generality and flexibility, we believe the choice of execution engine can become one based solely on the practical underlying differences between them: those of latency and resource cost.

Taken from that perspective, the conceptual contribution of this paper is a single unified model which:

- Allows for the calculation of event-time[5] ordered results, windowed by features of the data themselves, over an unbounded, unordered data source, with correctness, latency, and cost tunable across a broad spectrum of combinations.
- Decomposes pipeline implementation across four related dimensions, providing clarity, composability, and flexibility:
  - **What** results are being computed.
  - **Where** in event time they are being computed.
  - **When** in processing time they are materialized.
  - **How** earlier results relate to later refinements.
- Separates the logical notion of data processing from the underlying physical implementation, allowing the choice of batch, micro-batch, or streaming engine to become one of simply correctness, latency, and cost.

Concretely, this contribution is enabled by the following:

- A **windowing model** which supports unaligned event-time windows, and a simple API for their creation and use (Section 2.2).
- A **triggering model** that binds the output times of results to runtime characteristics of the pipeline, with a powerful and flexible declarative API for describing desired triggering semantics (Section 2.3).
- An **incremental processing model** that integrates retractions and updates into the windowing and triggering models described above (Section 2.3).
- **Scalable implementations** of the above atop the MillWheel streaming engine and the FlumeJava batch engine, with an external reimplementation for Google Cloud Dataflow, including an open-source SDK [19] that is runtime-agnostic (Section 3.1).
- A set of **core principles** that guided the design of this model (Section 3.2).
- Brief discussions of our **real-world experiences** with massive-scale, unbounded, out-of-order data processing at Google that motivated development of this model (Section 3.3).

It is lastly worth noting that there is nothing magical about this model. Things which are computationally impractical in existing strongly-consistent batch, micro-batch, streaming, or Lambda Architecture systems remain so, with the inherent constraints of CPU, RAM, and disk left steadfastly in place. What it does provide is a common framework

[2]By windowing, we mean as defined in Li [22], i.e. slicing data into finite chunks for processing. More in Section 1.2.
[3]By triggering, we mean stimulating the output of a specific window at a grouping operation. More in Section 2.3.
[4]By unaligned windows, we mean windows which do not span the entirety of a data source, but instead only a subset of it, such as per-user windows. This is essentially the frames idea from Whiteneck [31]. More in Section 1.2.
[5]By event times, we mean the times at which events occurred, not when they are processed. More in Section 1.3.

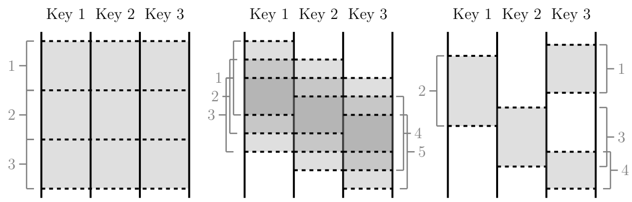

Figure 1: Common Windowing Patterns

that allows for the relatively simple expression of parallel computation in a way that is independent of the underlying execution engine, while also providing the ability to dial in precisely the amount of latency and correctness for any specific problem domain given the realities of the data and resources at hand. In that sense, it is a model aimed at ease of use in building practical, massive-scale data processing pipelines.

1.1 Unbounded/Bounded vs Streaming/Batch

When describing infinite/finite *data sets*, we prefer the terms unbounded/bounded over streaming/batch, because the latter terms carry with them an implication of the use of a specific type of execution engine. In reality, unbounded datasets have been processed using repeated runs of batch systems since their conception, and well-designed streaming systems are perfectly capable of processing bounded data. From the perspective of the model, the distinction of streaming or batch is largely irrelevant, and we thus reserve those terms exclusively for describing runtime *execution engines*.

1.2 Windowing

Windowing [22] slices up a dataset into finite chunks for processing as a group. When dealing with unbounded data, windowing is required for some operations (to delineate finite boundaries in most forms of grouping: aggregation, outer joins, time-bounded operations, etc.), and unnecessary for others (filtering, mapping, inner joins, etc.). For bounded data, windowing is essentially optional, though still a semantically useful concept in many situations (e.g. backfilling large scale updates to portions of a previously computed unbounded data source). Windowing is effectively always time based; while many systems support tuple-based windowing, this is essentially time-based windowing over a logical time domain where elements in order have successively increasing logical timestamps. Windows may be either *aligned*, i.e. applied across all the data for the window of time in question, or *unaligned*, i.e. applied across only specific subsets of the data (e.g. per key) for the given window of time. Figure 1 highlights three of the major types of windows encountered when dealing with unbounded data.

Fixed windows (sometimes called tumbling windows) are defined by a static window size, e.g. hourly windows or daily windows. They are generally aligned, i.e. every window applies across all of the data for the corresponding period of time. For the sake of spreading window completion load evenly across time, they are sometimes unaligned by phase shifting the windows for each key by some random value.

Sliding windows are defined by a window size and slide period, e.g. hourly windows starting every minute. The period may be less than the size, which means the windows may overlap. Sliding windows are also typically aligned; even though the diagram is drawn to give a sense of sliding motion, all five windows would be applied to all three keys in the diagram, not just Window 3. Fixed windows are really a special case of sliding windows where size equals period.

Sessions are windows that capture some period of activity over a subset of the data, in this case per key. Typically they are defined by a timeout gap. Any events that occur within a span of time less than the timeout are grouped together as a session. Sessions are unaligned windows. For example, Window 2 applies to Key 1 only, Window 3 to Key 2 only, and Windows 1 and 4 to Key 3 only.

1.3 Time Domains

When processing data which relate to events in time, there are two inherent domains of time to consider. Though captured in various places across the literature (particularly time management [28] and semantic models [9], but also windowing [22], out-of-order processing [23], punctuations [30], heartbeats [21], watermarks [2], frames [31]), the detailed examples in section 2.3 will be easier to follow with the concepts clearly in mind. The two domains of interest are:

- **Event Time**, which is the time at which the event itself actually *occurred*, i.e. a record of system clock time (for whatever system generated the event) at the time of occurrence.

- **Processing Time**, which is the time at which an event is observed at any given point *during processing* within the pipeline, i.e. the current time according to the system clock. Note that we make no assumptions about clock synchronization within a distributed system.

Event time for a given event essentially never changes, but processing time changes constantly for each event as it flows through the pipeline and time marches ever forward. This is an important distinction when it comes to robustly analyzing events in the context of when they occurred.

During processing, the realities of the systems in use (communication delays, scheduling algorithms, time spent processing, pipeline serialization, etc.) result in an inherent and dynamically changing amount of skew between the two domains. Global progress metrics, such as punctuations or watermarks, provide a good way to visualize this skew. For our purposes, we'll consider something like MillWheel's *watermark*, which is a lower bound (often heuristically established[6]) on event times that have been processed by the

[6]For most real-world distributed data sets, the system lacks sufficient knowledge to establish a 100% correct watermark. For example, in the video sessions use case, consider offline views. If someone takes their mobile device into the wilderness, the system has no practical way of knowing when they might come back to civilization, regain connection, and begin uploading data about video views during that time. As a result, most watermarks must be heuristically defined based on limited knowledge available. For structured input sources that expose metadata regarding unobserved data, such as log files, we've found these heuristics to be remarkably accurate, and thus practically useful as a completion estimate for many use cases. Furthermore, and importantly, once a heuristic watermark has been established, it can then be

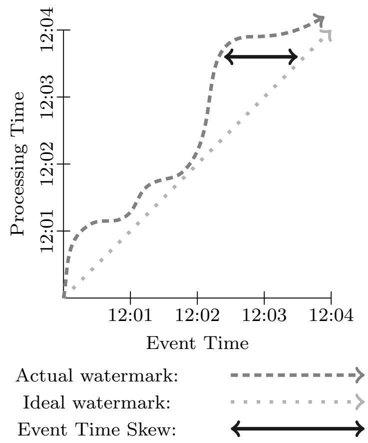

Figure 2: Time Domain Skew

pipeline. As we've made very clear above, notions of completeness are generally incompatible with correctness, so we won't rely on watermarks as such. They do, however, provide a useful notion of when the system thinks it likely that all data up to a given point in event time have been observed, and thus find application in not only visualizing skew, but in monitoring overall system health and progress, as well as making decisions around progress that do not require complete accuracy, such as basic garbage collection policies.

In an ideal world, time domain skew would always be zero; we would always be processing all events immediately as they happen. Reality is not so favorable, however, and often what we end up with looks more like Figure 2. Starting around 12:00, the watermark starts to skew more away from real time as the pipeline lags, diving back close to real time around 12:02, then lagging behind again noticeably by the time 12:03 rolls around. This dynamic variance in skew is very common in distributed data processing systems, and will play a big role in defining what functionality is necessary for providing correct, repeatable results.

2. DATAFLOW MODEL

In this section, we will define the formal model for the system and explain why its semantics are general enough to subsume the standard batch, micro-batch, and streaming models, as well as the hybrid streaming and batch semantics of the Lambda Architecture. For code examples, we will use a simplified variant of the Dataflow Java SDK, which itself is an evolution of the FlumeJava API.

2.1 Core Primitives

To begin with, let us consider primitives from the classic batch model. The Dataflow SDK has two core transforms that operate on the $(key, value)$ pairs flowing through the system[7]:

- **ParDo** for generic parallel processing. Each input element to be processed (which itself may be a finite collection) is provided to a user-defined function (called a `DoFn` in Dataflow), which can yield zero or more output elements per input. For example, consider an operation which expands all prefixes of the input key, duplicating the value across them:

$$(fix, 1), (fit, 2)$$

$$\Big\downarrow \; ParDo(ExpandPrefixes)$$

$$(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)$$

- ***GroupByKey*** for key-grouping $(key, value)$ pairs.

$$(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)$$

$$\Big\downarrow \; GroupByKey$$

$$(f, [1, 2]), (fi, [1, 2]), (fix, [1]), (fit, [2])$$

The ParDo operation operates element-wise on each input element, and thus translates naturally to unbounded data. The *GroupByKey* operation, on the other hand, collects *all* data for a given key before sending them downstream for reduction. If the input source is unbounded, we have no way of knowing when it will end. The common solution to this problem is to window the data.

2.2 Windowing

Systems which support grouping typically redefine their *GroupByKey* operation to essentially be *GroupByKeyAndWindow*. Our primary contribution here is support for unaligned windows, for which there are two key insights. The first is that it is simpler to treat all windowing strategies as unaligned from the perspective of the model, and allow underlying implementations to apply optimizations relevant to the aligned cases where applicable. The second is that windowing can be broken apart into two related operations:

- `Set<Window>` **`AssignWindows`**`(T datum)`, which assigns the element to zero or more windows. This is essentially the Bucket Operator from Li [22].
- `Set<Window>` **`MergeWindows`**`(Set<Window> windows)`, which merges windows at grouping time. This allows data-driven windows to be constructed over time as data arrive and are grouped together.

For any given windowing strategy, the two operations are intimately related; sliding window assignment requires sliding window merging, sessions window assignment requires sessions window merging, etc.

Note that, to support event-time windowing natively, instead of passing $(key, value)$ pairs through the system, we now pass (*key*, *value*, *event_time*, *window*) 4-tuples. Elements are provided to the system with event-time timestamps (which may also be modified at any point in the pipeline[8]), and are initially assigned to a default global window, covering all of event time, providing semantics that match the defaults in the standard batch model.

propagated accurately downstream through the rest of the pipeline (much like a punctuation would), though the overall metric itself remains a heuristic.

[7]Without loss of generality, we will treat all elements in the system as $(key, value)$ pairs, even though a key is not actually required for certain operations, such as ParDo. Most of the interesting discussions revolve around *GroupByKey*, which does require keys, so assuming they exist is simpler.

[8]Note, however, that certain timestamp modification operations are antagonistic to progress tracking metrics like watermarks; moving a timestamp behind the watermark makes a given element late with respect to that watermark.

$$(k, v_1, 12{:}00, [0, \infty)), (k, v_2, 12{:}01, [0, \infty))$$

$$\Big\downarrow \; \begin{array}{l} AssignWindows(\\ \quad Sliding(2m, 1m)) \end{array}$$

$$\begin{array}{c} (k, v_1, 12{:}00, [11{:}59, 12{:}01)), \\ (k, v_1, 12{:}00, [12{:}00, 12{:}02)), \\ (k, v_2, 12{:}01, [12{:}00, 12{:}02)), \\ (k, v_2, 12{:}01, [12{:}01, 12{:}03)) \end{array}$$

Figure 3: Window Assignment

2.2.1 Window Assignment

From the model's perspective, window assignment creates a new copy of the element in each of the windows to which it has been assigned. For example, consider windowing a dataset by sliding windows of two-minute width and one-minute period, as shown in Figure 3 (for brevity, timestamps are given in HH:MM format).

In this case, each of the two $(key, value)$ pairs is duplicated to exist in both of the windows that overlapped the element's timestamp. Since windows are associated directly with the elements to which they belong, this means window assignment can happen anywhere in the pipeline before grouping is applied. This is important, as the grouping operation may be buried somewhere downstream inside a composite transformation (e.g. `Sum.integersPerKey()`).

2.2.2 Window Merging

Window merging occurs as part of the *GroupByKeyAndWindow* operation, and is best explained in the context of an example. We will use session windowing since it is our motivating use case. Figure 4 shows four example data, three for k_1 and one for k_2, as they are windowed by session, with a 30-minute session timeout. All are initially placed in a default global window by the system. The sessions implementation of *AssignWindows* puts each element into a single window that extends 30 minutes beyond its own timestamp; this window denotes the range of time into which later events can fall if they are to be considered part of the same session. We then begin the *GroupByKeyAndWindow* operation, which is really a five-part composite operation:

- ***DropTimestamps*** - Drops element timestamps, as only the window is relevant from here on out[9].
- ***GroupByKey*** - Groups $(value, window)$ tuples by key.
- ***MergeWindows*** - Merges the set of currently buffered windows for a key. The actual merge logic is defined by the windowing strategy. In this case, the windows for v_1 and v_4 overlap, so the sessions windowing strategy merges them into a single new, larger session, as indicated in bold.
- ***GroupAlsoByWindow*** - For each key, groups values by window. After merging in the prior step, v_1 and v_4 are now in identical windows, and thus are grouped together at this step.

[9]If the user needs them later, it is possible to first materialize them as part of their value.

$$\begin{array}{c} (k_1, v_1, 13{:}02, [0, \infty)), \\ (k_2, v_2, 13{:}14, [0, \infty)), \\ (k_1, v_3, 13{:}57, [0, \infty)), \\ (k_1, v_4, 13{:}20, [0, \infty)) \end{array}$$

$$\Big\downarrow \; \begin{array}{l} AssignWindows(\\ \quad Sessions(30m)) \end{array}$$

$$\begin{array}{c} (k_1, v_1, 13{:}02, [13{:}02, 13{:}32)), \\ (k_2, v_2, 13{:}14, [13{:}14, 13{:}44)), \\ (k_1, v_3, 13{:}57, [13{:}57, 14{:}27)), \\ (k_1, v_4, 13{:}20, [13{:}20, 13{:}50)) \end{array}$$

$$\downarrow \; DropTimestamps$$

$$\begin{array}{c} (k_1, v_1, [13{:}02, 13{:}32)), \\ (k_2, v_2, [13{:}14, 13{:}44)), \\ (k_1, v_3, [13{:}57, 14{:}27)), \\ (k_1, v_4, [13{:}20, 13{:}50)) \end{array}$$

$$\downarrow \; GroupByKey$$

$$\begin{array}{l} (k_1, [(v_1, [13{:}02, 13{:}32)), \\ \qquad (v_3, [13{:}57, 14{:}27)), \\ \qquad (v_4, [13{:}20, 13{:}50))]), \\ (k_2, [(v_2, [13{:}14, 13{:}44))]) \end{array}$$

$$\Big\downarrow \; \begin{array}{l} MergeWindows(\\ \quad Sessions(30m)) \end{array}$$

$$\begin{array}{l} (k_1, [(v_1, [\mathbf{13{:}02}, \mathbf{13{:}50})), \\ \qquad (v_3, [13{:}57, 14{:}27)), \\ \qquad (v_4, [\mathbf{13{:}02}, \mathbf{13{:}50}))]), \\ (k_2, [(v_2, [13{:}14, 13{:}44))]) \end{array}$$

$$\downarrow \; GroupAlsoByWindow$$

$$\begin{array}{l} (k_1, [([\mathbf{v_1}, \mathbf{v_4}], [13{:}02, 13{:}50)), \\ \qquad ([\mathbf{v_3}], [13{:}57, 14{:}27))]), \\ (k_2, [([\mathbf{v_2}], [13{:}14, 13{:}44))]) \end{array}$$

$$\downarrow \; ExpandToElements$$

$$\begin{array}{c} (k_1, [v_1, v_4], \mathbf{13{:}50}, [13{:}02, 13{:}50)), \\ (k_1, [v_3], \mathbf{14{:}27}, [13{:}57, 14{:}27)), \\ (k_2, [v_2], \mathbf{13{:}44}, [13{:}14, 13{:}44)) \end{array}$$

Figure 4: Window Merging

- ***ExpandToElements*** - Expands per-key, per-window groups of values into $(key, value, event\_time, window)$ tuples, with new per-window timestamps. In this example, we set the timestamp to the end of the window, but any timestamp greater than or equal to the timestamp of the earliest event in the window is valid with respect to watermark correctness.

2.2.3 API

As a brief example of the use of windowing in practice, consider the following Cloud Dataflow SDK code to calculate keyed integer sums:

```
PCollection<KV<String, Integer>> input = IO.read(...);
PCollection<KV<String, Integer>> output = input
  .apply(Sum.integersPerKey());
```

To do the same thing, but windowed into sessions with a 30-minute timeout as in Figure 4, one would add a single `Window.into` call before initiating the summation:

```
PCollection<KV<String, Integer>> input = IO.read(...);
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(Sessions.withGapDuration(
      Duration.standardMinutes(30))))
  .apply(Sum.integersPerKey());
```

2.3 Triggers & Incremental Processing

The ability to build unaligned, event-time windows is an improvement, but now we have two more shortcomings to address:

- We need some way of providing support for tuple- and processing-time-based windows, otherwise we have regressed our windowing semantics relative to other systems in existence.
- We need some way of knowing when to emit the results for a window. Since the data are unordered with respect to event time, we require some other signal to tell us when the window is done.

The problem of tuple- and processing-time-based windows we will address in Section 2.4, once we have built up a solution to the window completeness problem. As to window completeness, an initial inclination for solving it might be to use some sort of global event-time progress metric, such as watermarks. However, watermarks themselves have two major shortcomings with respect to correctness:

- They are sometimes **too fast**, meaning there may be late data that arrives behind the watermark. For many distributed data sources, it is intractable to derive a completely perfect event time watermark, and thus impossible to rely on it solely if we want 100% correctness in our output data.
- They are sometimes **too slow**. Because they are a global progress metric, the watermark can be held back for the entire pipeline by a single slow datum. And even for healthy pipelines with little variability in event-time skew, the baseline level of skew may still be multiple minutes or more, depending upon the input source. As a result, using watermarks as the sole signal for emitting window results is likely to yield higher latency of overall results than, for example, a comparable Lambda Architecture pipeline.

For these reasons, we postulate that watermarks alone are insufficient. A useful insight in addressing the completeness problem is that the Lambda Architecture effectively sidesteps the issue: it does not solve the completeness problem by somehow providing correct answers faster; it simply provides the best low-latency estimate of a result that the streaming pipeline can provide, with the promise of eventual consistency and correctness once the batch pipeline runs[10]. If we want to do the same thing from within a single pipeline (regardless of execution engine), then we will need a way to provide multiple answers (or panes) for any given window. We call this feature triggers, since they allow the specification of when to trigger the output results for a given window.

In a nutshell, triggers are a mechanism for stimulating the production of *GroupByKeyAndWindow* results in response to internal or external signals. They are complementary to the windowing model, in that they each affect system behaviour along a different axis of time:

- **Windowing** determines *where* in **event time** data are grouped together for processing.
- **Triggering** determines *when* in **processing time** the results of groupings are emitted as panes.[11]

Our systems provide predefined trigger implementations for triggering at completion estimates (e.g. watermarks, including percentile watermarks, which provide useful semantics for dealing with stragglers in both batch and streaming execution engines when you care more about processing a minimum percentage of the input data quickly than processing every last piece of it), at points in processing time, and in response to data arriving (counts, bytes, data punctuations, pattern matching, etc.). We also support composing triggers into logical combinations (and, or, etc.), loops, sequences, and other such constructions. In addition, users may define their own triggers utilizing both the underlying primitives of the execution runtime (e.g. watermark timers, processing-time timers, data arrival, composition support) and any other relevant external signals (data injection requests, external progress metrics, RPC completion callbacks, etc.). We will look more closely at examples in Section 2.4.

In addition to controlling when results are emitted, the triggers system provides a way to control how multiple panes for the same window relate to each other, via three different refinement modes:

- **Discarding:** Upon triggering, window contents are discarded, and later results bear no relation to previous results. This mode is useful in cases where the downstream consumer of the data (either internal or external to the pipeline) expects the values from various trigger fires to be independent (e.g. when injecting into a system that generates a sum of the values injected). It is also the most efficient in terms of amount of data buffered, though for associative and commutative operations which can be modeled as a Dataflow `Combiner`, the efficiency delta will often be minimal. For our video sessions use case, this is not sufficient, since it is impractical to require downstream consumers of our data to stitch together partial sessions.
- **Accumulating**: Upon triggering, window contents are left intact in persistent state, and later results become a refinement of previous results. This is useful when the downstream consumer expects to overwrite old values with new ones when receiving multiple results for the same window, and is effectively the mode used in Lambda Architecture systems, where the

[10] Note that in reality, output from the batch job is only correct if input data is complete by the time the batch job runs; if data evolve over time, this must be detected and the batch jobs re-executed.

[11] Specific triggers, such as watermark triggers, make use of event time in the functionality they provide, but their effects within the pipeline are still realized in the processing time axis.

streaming pipeline produces low-latency results, which are then overwritten in the future by the results from the batch pipeline. For video sessions, this might be sufficient if we are simply calculating sessions and then immediately writing them to some output source that supports updates (e.g. a database or key/value store).

- **Accumulating & Retracting**: Upon triggering, in addition to the *Accumulating* semantics, a copy of the emitted value is also stored in persistent state. When the window triggers again in the future, a retraction for the previous value will be emitted first, followed by the new value as a normal datum[12]. Retractions are necessary in pipelines with multiple serial *GroupByKeyAndWindow* operations, since the multiple results generated by a single window over subsequent trigger fires may end up on separate keys when grouped downstream. In that case, the second grouping operation will generate incorrect results for those keys unless it is informed via a retraction that the effects of the original output should be reversed. Dataflow `Combiner` operations that are also reversible can support retractions efficiently via an `uncombine` method. For video sessions, this mode is the ideal. If we are performing aggregations downstream from session creation that depend on properties of the sessions themselves, for example detecting unpopular ads (such as those which are viewed for less than five seconds in a majority of sessions), initial results may be invalidated as inputs evolve over time, e.g. as a significant number of offline mobile viewers come back online and upload session data. Retractions provide a way for us to adapt to these types of changes in complex pipelines with multiple serial grouping stages.

2.4 Examples

We will now consider a series of examples that highlight the plurality of useful output patterns supported by the Dataflow Model. We will look at each example in the context of the integer summation pipeline from Section 2.2.3:

```
PCollection<KV<String, Integer>> output = input
  .apply(Sum.integersPerKey());
```

Let us assume we have an input source from which we are observing ten data points, each themselves small integer values. We will consider them in the context of both bounded and unbounded data sources. For diagrammatic simplicity, we will assume all these data are for the same key; in a real pipeline, the types of operations we describe here would be happening in parallel for multiple keys. Figure 5 diagrams how these data relate together along both axes of time we care about. The X axis plots the data in event time (i.e. when the events actually occurred), while the Y axis plots the data in processing time (i.e. when the pipeline observes them). All examples assume execution on our streaming engine unless otherwise specified.

Many of the examples will also depend on watermarks, in which cases we will include them in our diagrams. We will graph both the ideal watermark and an example actual watermark. The straight dotted line with slope of one represents the ideal watermark, i.e. if there were no event-time skew and all events were processed by the system as they occurred. Given the vagaries of distributed systems, skew is a common occurrence; this is exemplified by the meandering path the actual watermark takes in Figure 5, represented by the darker, dashed line. Note also that the heuristic nature of this watermark is exemplified by the single "late" datum with value 9 that appears behind the watermark.

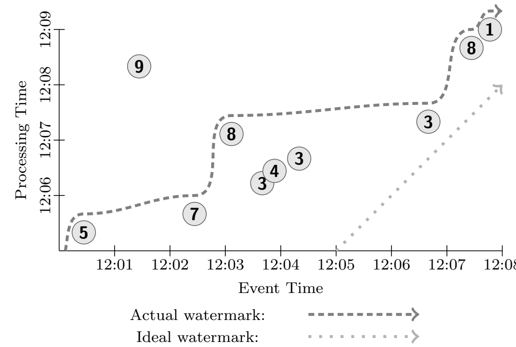

Figure 5: Example Inputs

If we were to process these data in a classic batch system using the described summation pipeline, we would wait for all the data to arrive, group them together into one bundle (since these data are all for the same key), and sum their values to arrive at total result of 51. This result is represented by the darkened rectangle in Figure 6, whose area covers the ranges of event and processing time included in the sum (with the top of the rectangle denoting when in processing time the result was materialized). Since classic batch processing is event-time agnostic, the result is contained within a single global window covering all of event time. And since outputs are only calculated once all inputs are received, the result covers all of processing time for the execution.

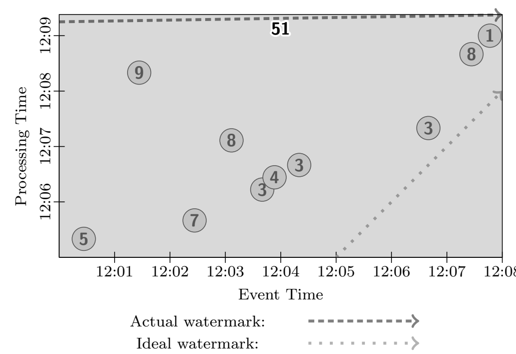

Figure 6: Classic Batch Execution

Note the inclusion of watermarks in this diagram. Though not typically used for classic batch processing, watermarks would semantically be held at the beginning of time until all data had been processed, then advanced to infinity. An important point to note is that one can get identical semantics to classic batch by running the data through a streaming system with watermarks progressed in this manner.

Now let us say we want to convert this pipeline to run over an unbounded data source. In Dataflow, the default triggering semantics are to emit windows when the watermark

[12]A simple implementation of retraction processing requires deterministic operations, but non-determinism may be supported with additional complexity and cost; we have seen use cases that require this, such as probabilistic modeling.

passes them. But when using the global window with an unbounded input source, we are guaranteed that will never happen, since the global window covers all of event time. As such, we will need to either trigger by something other than the default trigger, or window by something other than the global window. Otherwise, we will never get any output.

Let us first look at changing the trigger, since this will allow us to to generate conceptually identical output (a global per-key sum over all time), but with periodic updates. In this example, we apply a `Window.trigger` operation that repeatedly fires on one-minute periodic processing-time boundaries. We also specify *Accumulating* mode so that our global sum will be refined over time (this assumes we have an output sink into which we can simply overwrite previous results for the key with new results, e.g. a database or key/value store). Thus, in Figure 7, we generate updated global sums once per minute of processing time. Note how the semi-transparent output rectangles overlap, since *Accumulating* panes build upon prior results by incorporating overlapping regions of processing time:

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
               .accumulating())
  .apply(Sum.integersPerKey());
```

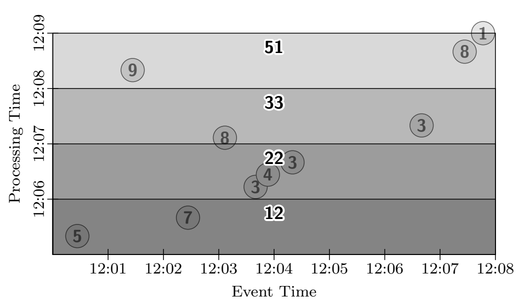

Figure 7: GlobalWindows, AtPeriod, Accumulating

If we instead wanted to generate the *delta* in sums once per minute, we could switch to *Discarding* mode, as in Figure 8. Note that this effectively gives the processing-time windowing semantics provided by many streaming systems. The output panes no longer overlap, since their results incorporate data from independent regions of processing time.

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
               .discarding())
  .apply(Sum.integersPerKey());
```

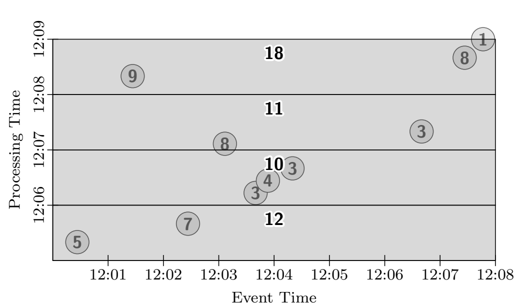

Figure 8: GlobalWindows, AtPeriod, Discarding

Another, more robust way of providing processing-time windowing semantics is to simply assign arrival time as event times at data ingress, then use event time windowing. A nice side effect of using arrival time event times is that the system has perfect knowledge of the event times in flight, and thus can provide perfect (i.e. non-heuristic) watermarks, with no late data. This is an effective and cost-efficient way of processing unbounded data for use cases where true event times are not necessary or available.

Before we look more closely at other windowing options, let us consider one more change to the triggers for this pipeline. The other common windowing mode we would like to model is tuple-based windows. We can provide this sort of functionality by simply changing the trigger to fire after a certain number of data arrive, say two. In Figure 9, we get five outputs, each containing the sum of two adjacent (by processing time) data. More sophisticated tuple-based windowing schemes (e.g. sliding tuple-based windows) require custom windowing strategies, but are otherwise supported.

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtCount(2)))
               .discarding())
  .apply(Sum.integersPerKey());
```

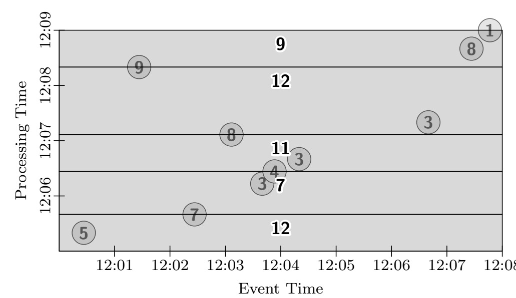

Figure 9: GlobalWindows, AtCount, Discarding

Let us now return to the other option for supporting unbounded sources: switching away from global windowing. To start with, let us window the data into fixed, two-minute *Accumulating* windows:

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES)
               .accumulating())
  .apply(Sum.integersPerKey());
```

With no trigger strategy specified, the system would use the default trigger, which is effectively:

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
               .trigger(Repeat(AtWatermark())))
               .accumulating())
  .apply(Sum.integersPerKey());
```

The watermark trigger fires when the watermark passes the end of the window in question. Both batch and streaming engines implement watermarks, as detailed in Section 3.1. The `Repeat` call in the trigger is used to handle late data; should any data arrive after the watermark, they will instantiate the repeated watermark trigger, which will fire immediately since the watermark has already passed.

Figures 10−12 each characterize this pipeline on a different type of runtime engine. We will first observe what execution of this pipeline would look like on a batch engine. Given our current implementation, the data source would have to be a bounded one, so as with the classic batch example above, we would wait for all data in the batch to arrive. We would then process the data in event-time order, with windows being emitted as the simulated watermark advances, as in Figure 10:

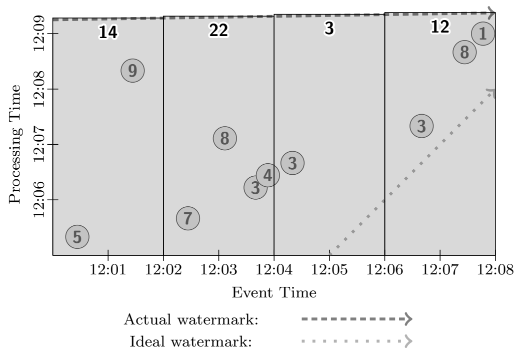

Figure 10: FixedWindows, Batch

Now imagine executing a micro-batch engine over this data source with one minute micro-batches. The system would gather input data for one minute, process them, and repeat. Each time, the watermark for the current batch would start at the beginning of time and advance to the end of time (technically jumping from the end time of the batch to the end of time instantaneously, since no data would exist for that period). We would thus end up with a new watermark for every micro-batch round, and corresponding outputs for all windows whose contents had changed since the last round. This provides a very nice mix of latency and eventual correctness, as in Figure 11:

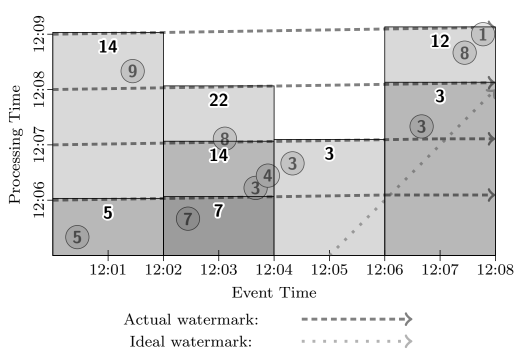

Figure 11: FixedWindows, Micro-Batch

Next, consider this pipeline executed on a streaming engine, as in Figure 12. Most windows are emitted when the watermark passes them. Note however that the datum with value 9 is actually late relative to the watermark. For whatever reason (mobile input source being offline, network partition, etc.), the system did not realize that datum had not yet been injected, and thus, having observed the 5, allowed the watermark to proceed past the point in event time that would eventually be occupied by the 9. Hence, once the 9 finally arrives, it causes the first window (for event-time range [12:00, 12:02)) to retrigger with an updated sum:

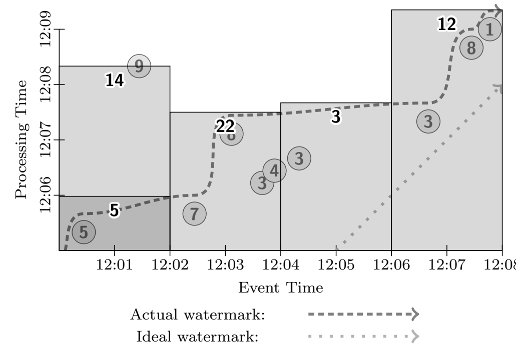

Figure 12: FixedWindows, Streaming

This output pattern is nice in that we have roughly one output per window, with a single refinement in the case of the late datum. But the overall latency of results is noticeably worse than the micro-batch system, on account of having to wait for the watermark to advance; this is the case of watermarks being too slow from Section 2.3.

If we want lower latency via multiple partial results for all of our windows, we can add in some additional, processing-time-based triggers to provide us with regular updates until the watermark actually passes, as in Figure 13. This yields somewhat better latency than the micro-batch pipeline, since data are accumulated in windows as they arrive instead of being processed in small batches. Given strongly-consistent micro-batch and streaming engines, the choice between them (as well as the choice of micro-batch size) really becomes just a matter of latency versus cost, which is exactly one of the goals we set out to achieve with this model.

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
               .trigger(SequenceOf(
                 RepeatUntil(
                   AtPeriod(1, MINUTE),
                   AtWatermark()),
                 Repeat(AtWatermark())))
               .accumulating())
  .apply(Sum.integersPerKey());
```

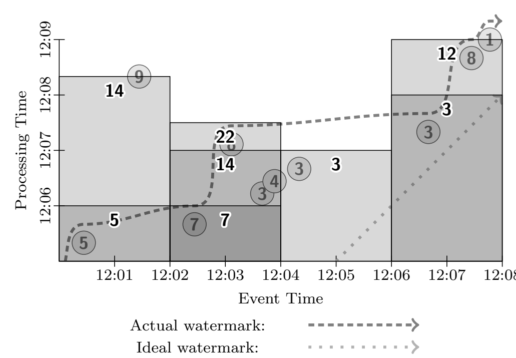

Figure 13: FixedWindows, Streaming, Partial

As one final exercise, let us update our example to satisfy the video sessions requirements (modulo the use of summation as the aggregation operation, which we will maintain for diagrammatic consistency; switching to another aggregation would be trivial), by updating to session windowing with a one minute timeout and enabling retractions. This highlights the composability provided by breaking the model into four pieces (*what* you are computing, *where* in event time you are computing it, *when* in processing time you are observing the answers, and *how* those answers relate to later refinements), and also illustrates the power of reverting previous values which otherwise might be left uncorrelated to the value offered as replacement.

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(Sessions.withGapDuration(1, MINUTE))
               .trigger(SequenceOf(
                 RepeatUntil(
                   AtPeriod(1, MINUTE),
                   AtWatermark()),
                 Repeat(AtWatermark())))
               .accumulatingAndRetracting())
  .apply(Sum.integersPerKey());
```

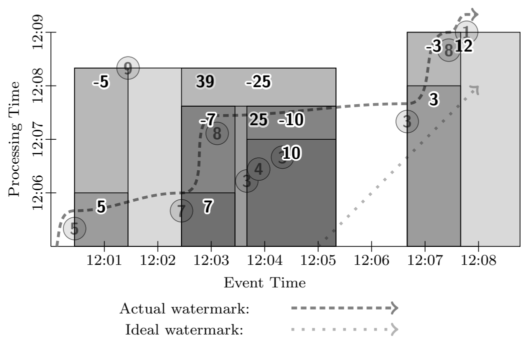

Figure 14: Sessions, Retracting

In this example, we output initial singleton sessions for values 5 and 7 at the first one-minute processing-time boundary. At the second minute boundary, we output a third session with value 10, built up from the values 3, 4, and 3. When the value of 8 is finally observed, it joins the two sessions with values 7 and 10. As the watermark passes the end of this new combined session, retractions for the 7 and 10 sessions are emitted, as well as a normal datum for the new session with value 25. Similarly, when the 9 arrives (late), it joins the session with value 5 to the session with value 25. The repeated watermark trigger then immediately emits retractions for the 5 and the 25, followed by a combined session of value 39. A similar dance occurs for the values 3, 8, and 1, ultimately ending with a retraction for an initial 3 session, followed by a combined session of 12.

3. IMPLEMENTATION & DESIGN

3.1 Implementation

We have implemented this model internally in FlumeJava, with MillWheel used as the underlying execution engine for streaming mode; additionally, an external reimplementation for Cloud Dataflow is largely complete at the time of writing. Due to prior characterization of those internal systems in the literature, as well as Cloud Dataflow being publicly available, details of the implementations themselves are elided here for the sake of brevity. One interesting note is that the core windowing and triggering code is quite general, and a significant portion of it is shared across batch and streaming implementations; that system itself is worthy of a more detailed analysis in future work.

3.2 Design Principles

Though much of our design was motivated by the real-world experiences detailed in Section 3.3 below, it was also guided by a core set of principles that we believed our model should embody:

- Never rely on any notion of completeness.
- Be flexible, to accommodate the diversity of known use cases, and those to come in the future.
- Not only make sense, but also add value, in the context of each of the envisioned execution engines.
- Encourage clarity of implementation.
- Support robust analysis of data in the context in which they occurred.

While the experiences below informed specific features of the model, these principles informed the overall shape and character of it, and we believe ultimately led to a more comprehensive and general result.

3.3 Motivating Experiences

As we designed the Dataflow Model, we took into consideration our real-world experiences with FlumeJava and MillWheel over the years. Things which worked well, we made sure to capture in the model; things which worked less well motivated changes in approach. Here are brief summaries of some of these experiences that influenced our design.

3.3.1 Large Scale Backfills & The Lambda Architecture: Unified Model

A number of teams run log joining pipelines on MillWheel. One particularly large log join pipeline runs in streaming mode on MillWheel by default, but has a separate FlumeJava batch implementation used for large scale backfills. A much nicer setup would be to have a single implementation written in a unified model that could run in both streaming and batch mode without modification. This became the initial motivating use case for unification across batch, micro-batch, and streaming engines, and was highlighted in Figures 10−12.

Another motivation for the unified model came from an experience with the Lambda Architecture. Though most data processing use cases at Google are handled exclusively by a batch or streaming system, one MillWheel customer ran their streaming pipeline in weak consistency mode, with a nightly MapReduce to generate truth. They found that customers stopped trusting the weakly consistent results over time, and as a result reimplemented their system around strong consistency so they could provide reliable, low latency results. This experience further motivated the desire to support fluid choice amongst execution engines.

3.3.2 Unaligned Windows: Sessions

From the outset, we knew we needed to support sessions; this in fact is the main contribution of our windowing model over existing models. Sessions are an extremely important use case within Google (and were in fact one of the reasons MillWheel was created), and are used across a number of product areas, including search, ads, analytics, social, and YouTube. Pretty much anyone that cares about correlating bursts of otherwise disjoint user activity over a period of time does so by calculating sessions. Thus, support for sessions became paramount in our design. As shown in Figure 14, generating sessions in the Dataflow Model is trivial.

3.3.3 Billing: Triggers, Accumulation, & Retraction

Two teams with billing pipelines built on MillWheel experienced issues that motivated parts of the model. Recommended practice at the time was to use the watermark as a completion metric, with ad hoc logic to deal with late data or changes in source data. Lacking a principled system for updates and retractions, a team that processed resource utilization statistics ended up leaving our platform to build a custom solution (the model for which ended being quite similar to the one we developed concurrently). Another billing team had significant issues with watermark lags caused by stragglers in their input. These shortcomings became major motivators in our design, and influenced the shift of focus from one of targeting completeness to one of adaptability over time. The results were twofold: triggers, which allow the concise and flexible specification of when results are materialized, as evidenced by the variety of output patterns possible over the same data set in Figures 7−14; and incremental processing support via accumulation (Figures 7 and 8) and retractions (Figure 14).

3.3.4 Statistics Calculation: Watermark Triggers

Many MillWheel pipelines calculate aggregate statistics (e.g. latency averages). For them, 100% accuracy is not required, but having a largely complete view of their data in a reasonable amount of time is. Given the high level of accuracy we achieve with watermarks for structured input sources like log files, such customers find watermarks very effective in triggering a single, highly-accurate aggregate per window. Watermark triggers are highlighted in Figure 12.

A number of abuse detection pipelines run on MillWheel. Abuse detection is another example of a use case where processing a majority of the data quickly is much more useful than processing 100% of the data more slowly. As such, they are heavy users of MillWheel's percentile watermarks, and were a strong motivating case for being able to support percentile watermark triggers in the model.

Relatedly, a pain point with batch processing jobs is stragglers that create a long tail in execution time. While dynamic rebalancing can help with this issue, FlumeJava has a custom feature that allows for early termination of a job based on overall progress. One of the benefits of the unified model for batch mode is that this sort of early termination criteria is now naturally expressible using the standard triggers mechanism, rather than requiring a custom feature.

3.3.5 Recommendations: Processing Time Triggers

Another pipeline that we considered built trees of user activity (essentially session trees) across a large Google property. These trees were then used to build recommendations tailored to users' interests. The pipeline was noteworthy in that it used processing-time timers to drive its output. This was due to the fact that, for their system, having regularly updated, partial views on the data was much more valuable than waiting until mostly complete views were ready once the watermark passed the end of the session. It also meant that lags in watermark progress due to a small amount of slow data would not affect timeliness of output for the rest of the data. This pipeline thus motivated inclusion of processing-time triggers shown in Figures 7 and 8.

3.3.6 Anomaly Detection: Data-Driven & Composite Triggers

In the MillWheel paper, we described an anomaly detection pipeline used to track trends in Google web search queries. When developing triggers, their diff detection system motivated data-driven triggers. These differs observe the stream of queries and calculate statistical estimates of whether a spike exists or not. When they believe a spike is happening, they emit a start record, and when they believe it has ceased, they emit a stop. Though you could drive the differ output with something periodic like Trill's punctuations, for anomaly detection you ideally want output as soon as you are confident you have discovered an anomaly; the use of punctuations essentially transforms the streaming system into micro-batch, introducing additional latency. While practical for a number of use cases, it ultimately is not an ideal fit for this one, thus motivating support for custom data-driven triggers. It was also a motivating case for trigger composition, because in reality, the system runs multiple differs at once, multiplexing the output of them according to a well-defined set of logic. The `AtCount` trigger used in Figure 9 exemplified data-driven triggers; figures 10−14 utilized composite triggers.

4. CONCLUSIONS

The future of data processing is unbounded data. Though bounded data will always have an important and useful place, it is semantically subsumed by its unbounded counterpart. Furthermore, the proliferation of unbounded data sets across modern business is staggering. At the same time, consumers of processed data grow savvier by the day, demanding powerful constructs like event-time ordering and unaligned windows. The models and systems that exist today serve as an excellent foundation on which to build the data processing tools of tomorrow, but we firmly believe that a shift in overall mindset is necessary to enable those tools to comprehensively address the needs of consumers of unbounded data.

Based on our many years of experience with real-world, massive-scale, unbounded data processing within Google, we believe the model presented here is a good step in that direction. It supports the unaligned, event-time-ordered windows modern data consumers require. It provides flexible triggering and integrated accumulation and retraction, refocusing the approach from one of finding completeness in data to one of adapting to the ever present changes manifest in real-world datasets. It abstracts away the distinction of batch vs. micro-batch vs. streaming, allowing pipeline builders a more fluid choice between them, while shielding them from the system-specific constructs that inevitably creep into models targeted at a single underlying system. Its overall flexibility allows pipeline builders to appropriately balance the dimensions of correctness, latency, and cost to fit their use case, which is critical given the diversity of needs in existence. And lastly, it clarifies pipeline implementations by separating the notions of what results are being computed, where in event time they are being computed, when in processing time they are materialized, and how earlier results relate to later refinements. We hope others will find this model useful as we all continue to push forward the state of the art in this fascinating, remarkably complex field.

5. ACKNOWLEDGMENTS

We thank all of our faithful reviewers for their dedication, time, and thoughtful comments: Atul Adya, Ben Birt, Ben Chambers, Cosmin Arad, Matt Austern, Lukasz Cwik, Grzegorz Czajkowski, Walt Drummond, Jeff Gardner, Anthony Mancuso, Colin Meek, Daniel Myers, Sunil Pedapudi, Amy Unruh, and William Vambenepe. We also wish to recognize the impressive and tireless efforts of everyone on the Google Cloud Dataflow, FlumeJava, MillWheel, and related teams that have helped bring this work to life.

6. REFERENCES

[1] D. J. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] T. Akidau et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Proc. of the 39th Int. Conf. on Very Large Data Bases (VLDB)*, 2013.

[3] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[4] Apache. Apache Hadoop. `http://hadoop.apache.org`, 2012.

[5] Apache. Apache Storm. `http://storm.apache.org`, 2013.

[6] Apache. Apache Flink. `http://flink.apache.org/`, 2014.

[7] Apache. Apache Samza. `http://samza.apache.org`, 2014.

[8] R. S. Barga et al. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Proc. of the Third Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 363–374, 2007.

[9] Botan et al. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. VLDB Endow.*, 3(1-2):232–243, Sept. 2010.

[10] O. Boykin et al. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proc. VLDB Endow.*, 7(13):1441–1451, Aug. 2014.

[11] Cask. Tigon. `http://tigon.io/`, 2015.

[12] C. Chambers et al. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[13] B. Chandramouli et al. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In *Proc. of the 41st Int. Conf. on Very Large Data Bases (VLDB)*, 2015.

[14] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[15] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the 2000 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, pages 379–390, 2000.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[17] EsperTech. Esper. `http://www.espertech.com/esper/`, 2006.

[18] Gates et al. Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.

[19] Google. Dataflow SDK. `https://github.com/GoogleCloudPlatform/DataflowJavaSDK`, 2015.

[20] Google. Google Cloud Dataflow. `https://cloud.google.com/dataflow/`, 2015.

[21] T. Johnson et al. A Heartbeat Mechanism and its Application in Gigascope. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB)*, pages 1079–1088, 2005.

[22] J. Li et al. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings og the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, pages 311–322, 2005.

[23] J. Li et al. Out-of-order Processing: A New Architecture for High-performance Stream Systems. *Proc. VLDB Endow.*, 1(1):274–288, Aug. 2008.

[24] D. Maier et al. Semantics of Data Streams and Operators. In *Proc. of the 10th Int. Conf. on Database Theory (ICDT)*, pages 37–52, 2005.

[25] N. Marz. How to beat the CAP theorem. `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`, 2011.

[26] S. Murthy et al. Pulsar – Real-Time Analytics at Scale. Technical report, eBay, 2015.

[27] SQLStream. `http://sqlstream.com/`, 2015.

[28] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *Proc. of the 23rd ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Systems*, pages 263–274, 2004.

[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[30] P. A. Tucker et al. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15, 2003.

[31] J. Whiteneck et al. Framing the Question: Detecting and Filling Spatial- Temporal Windows. In *Proc. of the ACM SIGSPATIAL Int. Workshop on GeoStreaming (IWGS)*, 2010.

[32] F. Yang and others. Sonora: A Platform for Continuous Mobile-Cloud Computing. Technical Report MSR-TR-2012-34, Microsoft Research Asia.

[33] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.

[34] M. Zaharia et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. of the 24th ACM Symp. on Operating Systems Principles*, 2013.