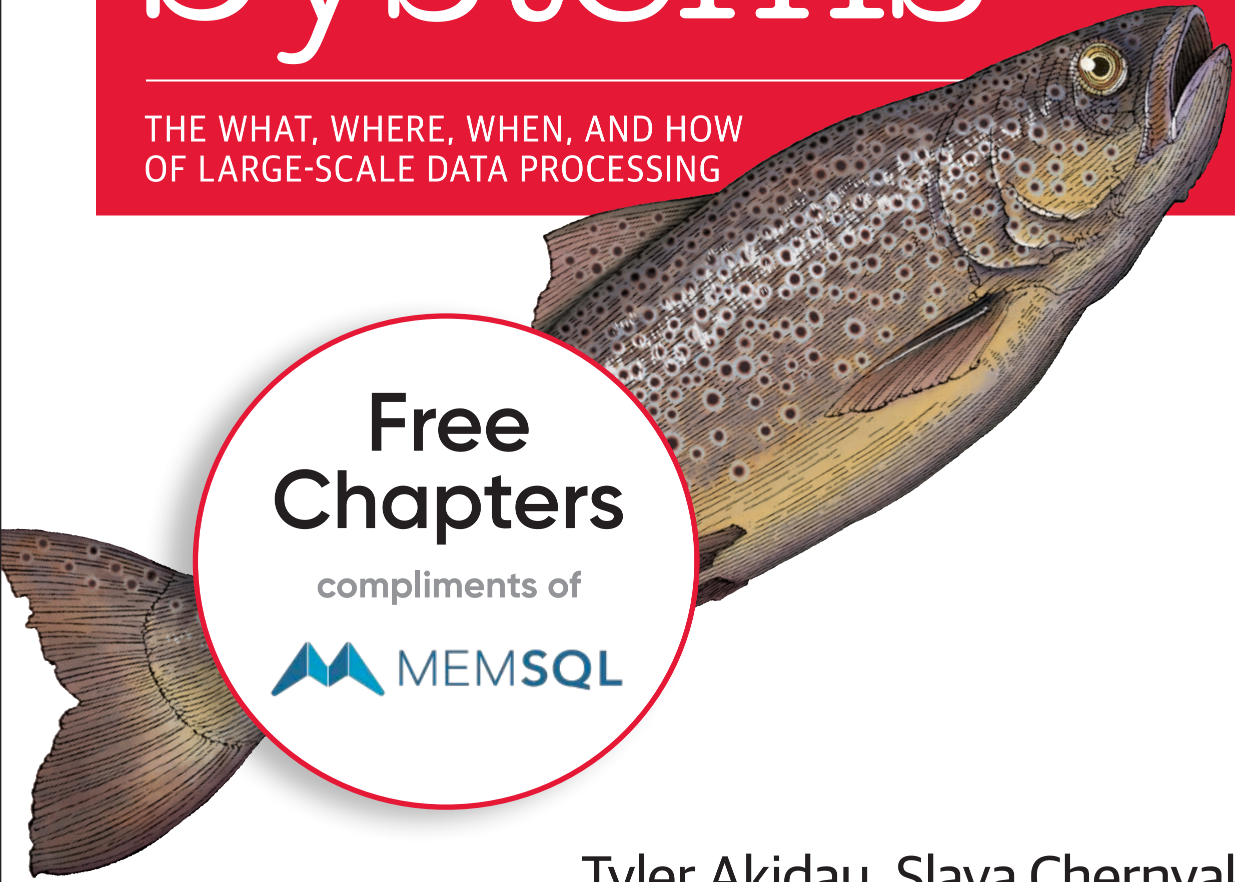
O'REILLY®

Streaming Systems

THE WHAT, WHERE, WHEN, AND HOW OF LARGE-SCALE DATA PROCESSING



Free Chapters

compliments of

MEMSQL

Tyler Akidau, Slava Chernyak & Reuven Lax

MEMSQL

THE NO-LIMITS DATABASE™

For your most demanding **Data Streaming** workloads

No Compromise
Familiar SQL with time series data

No Latency
Fast ingest, fast insights

No Lock-In
Public, private, & hybrid cloud

Try it FREE today at **MemSQL.com**

Streaming Systems

The What, Where, When, and How of Large-Scale Data Processing

Tyler Akidau, Slava Chernyak, and Reuven Lax

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

Streaming Systems
by Tyler Akidau, Slava Chernyak, and Reuven Lax

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Rachel Roumeliotis and Jeff Bleiel
Production Editor: Nicholas Adams
Copyeditor: Octal Publishing, Inc.
Proofreader: Kim Cofer
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2018: First Edition

Revision History for the First Edition
2018-07-12: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491983874* for release details.

978-1-491-98387-4

[LSI]

Table of Contents

Foreword

Interest in streaming is growing rapidly. Running operations globally and 24/7 is becoming the norm. Sensors and mobile phones are always on, constantly delivering live data that needs a response. Operating in real time offers businesses the opportunity to improve the customer's experience and gain competitive advantage.

Also, recently, operating in real time has moved from impossible, to hard and expensive, and now to achievable. It's little wonder that streaming data is part of the biggest new trends in information, including the Internet of Things, machine learning, and artificial intelligence.

In *In Streaming Systems: The What, When, Where, and How of Large-Scale Data Processing*, authors Tyler Akidau, Slava Chernyak, and Reuven Lax tell you how to extract value from streaming data. They set the stage with careful definitions of streaming data and the role of streaming in every step of data processing. They then take a deep dive into important technical issues such as windows on data, exactly-once processing (supported by both Kafka and MemSQL), how streams interact with data tables, and the usefulness of SQL in a streaming architecture. They finish by discussing the use of joins with streaming data and how streaming is revolutionizing data processing as a whole. In this free excerpt, we feature two key chapters from *Streaming Systems*:

Streaming 101 (Chapter 1)
: What is—and isn't—streaming data, why there are few actual limitations on streaming, and an emphasis on the role of unbounded data in streaming systems.

The What, Where, When, and How of Data Processing (Chapter 2)
: Describes data processing's foundations in the batch processing world and its evolution in the streaming world, relating streaming to data's increased value

We here at MemSQL are proud to offer you this free excerpt, which may help you unlock the full potential of the data available to your business. Traditional databases are not designed to efficiently and scalably power streaming data applications.

MemSQL, by contrast, provides a modern, distributed lock-free architecture that pairs perfectly with streaming technologies such as Kafka and Spark. The combined solution delivers rapid ingest, fast transaction and immediate analytical processing, using familiar SQL.

We hope you enjoy this excerpt from *Streaming Systems*. We encourage you to put the information to use in your own applications.

— Peter Guagenti
MemSQL
April 2019

CHAPTER 1

Streaming 101

Streaming data processing is a big deal in big data these days, and for good reasons; among them are the following:

- Businesses crave ever-more timely insights into their data, and switching to streaming is a good way to achieve lower latency
- The massive, unbounded datasets that are increasingly common in modern business are more easily tamed using a system designed for such never-ending volumes of data.
- Processing data as they arrive spreads workloads out more evenly over time, yielding more consistent and predictable consumption of resources.

Despite this business-driven surge of interest in streaming, streaming systems long remained relatively immature compared to their batch brethren. It's only recently that the tide has swung conclusively in the other direction. In my more bumptious moments, I hope that might be in small part due to the solid dose of goading I originally served up in my "Streaming 101" and "Streaming 102" blog posts (on which the first few chapters of this book are rather obviously based). But in reality, there's also just a lot of industry interest in seeing streaming systems mature and a lot of smart and active folks out there who enjoy building them.

Even though the battle for general streaming advocacy has been, in my opinion, effectively won, I'm still going to present my original arguments from "Streaming 101" more or less unaltered. For one, they're still very applicable today, even if much of industry has begun to heed the battle cry. And for two, there are a lot of folks out there who still haven't gotten the memo; this book is an extended attempt at getting these points across.

To begin, I cover some important background information that will help frame the rest of the topics I want to discuss. I do this in three specific sections:

Terminology
: To talk precisely about complex topics requires precise definitions of terms. For some terms that have overloaded interpretations in current use, I'll try to nail down exactly what I mean when I say them.

Capabilities
: I remark on the oft-perceived shortcomings of streaming systems. I also propose the frame of mind that I believe data processing system builders need to adopt in order to address the needs of modern data consumers going forward.

Time domains
: I introduce the two primary domains of time that are relevant in data processing, show how they relate, and point out some of the difficulties these two domains impose.

Terminology: What Is Streaming?

Before going any further, I'd like to get one thing out of the way: what is streaming? The term streaming is used today to mean a variety of different things (and for simplicity I've been using it somewhat loosely up until now), which can lead to misunderstandings about what streaming really is or what streaming systems are actually capable of. As a result, I would prefer to define the term somewhat precisely.

The crux of the problem is that many things that ought to be described by *what* they are (unbounded data processing, approximate results, etc.), have come to be described colloquially by *how* they historically have been accomplished (i.e., via streaming execution engines). This lack of precision in terminology clouds what streaming really means, and in some cases it burdens streaming systems themselves with the implication that their capabilities are limited to characteristics historically described as "streaming," such as approximate or speculative results.

Given that well-designed streaming systems are just as capable (technically more so) of producing correct, consistent, repeatable results as any existing batch engine, I prefer to isolate the term "streaming" to a very specific meaning:

Streaming system
: A type of data processing engine that is designed with infinite datasets in mind.[1]

1 For completeness, it's perhaps worth calling out that this definition includes both true streaming as well as microbatch implementations. For those of you who aren't familiar with microbatch systems, they are streaming systems that use repeated executions of a batch processing engine to process unbounded data. Spark Streaming is the canonical example in the industry.

If I want to talk about low-latency, approximate, or speculative results, I use those specific words rather than imprecisely calling them "streaming."

Precise terms are also useful when discussing the different types of data one might encounter. From my perspective, there are two important (and orthogonal) dimensions that define the shape of a given dataset: *cardinality* and *constitution*.

The cardinality of a dataset dictates its size, with the most salient aspect of cardinality being whether a given dataset is finite or infinite. Here are the two terms I prefer to use for describing the coarse cardinality in a dataset:

Bounded data
: A type of dataset that is finite in size.

Unbounded data
: A type of dataset that is infinite in size (at least theoretically).

Cardinality is important because the unbounded nature of infinite datasets imposes additional burdens on data processing frameworks that consume them. More on this in the next section.

The constitution of a dataset, on the other hand, dictates its physical manifestation. As a result, the constitution defines the ways one can interact with the data in question. We won't get around to deeply examining constitutions until later in this book, but to give you a brief sense of things, there are two primary constitutions of importance:

Table
: A holistic view of a dataset at a specific point in time. SQL systems have traditionally dealt in tables.

Stream[2]
: An element-by-element view of the evolution of a dataset over time. The MapReduce lineage of data processing systems have traditionally dealt in streams.

We look quite deeply at the relationship between streams and tables later in this book, and we'll also learn about the unifying underlying concept of *time-varying relations* that ties them together. But until then, we deal primarily in streams because that's the constitution pipeline developers directly interact with in most data processing sys-

2 Readers familiar with my original "Streaming 101" article might recall that I rather emphatically encouraged the abandonment of the term "stream" when referring to datasets. That never caught on, which I initially thought was due to its catchiness and pervasive existing usage. In retrospect, however, I think I was simply wrong. There actually is great value in distinguishing between the two different types of dataset constitutions: tables and streams. Indeed, most of the second half of this book is dedicated to understanding the relationship between those two.

tems today (both batch and streaming). It's also the constitution that most naturally embodies the challenges that are unique to stream processing.

On the Greatly Exaggerated Limitations of Streaming

On that note, let's next talk a bit about what streaming systems can and can't do, with an emphasis on can. One of the biggest things I want to get across in this chapter is just how capable a well-designed streaming system can be. Streaming systems have historically been relegated to a somewhat niche market of providing low-latency, inaccurate, or speculative results, often in conjunction with a more capable batch system to provide eventually correct results; in other words, the Lambda Architecture.

For those of you not already familiar with the Lambda Architecture, the basic idea is that you run a streaming system alongside a batch system, both performing essentially the same calculation. The streaming system gives you low-latency, inaccurate results (either because of the use of an approximation algorithm, or because the streaming system itself does not provide correctness), and some time later a batch system rolls along and provides you with correct output. Originally proposed by Twitter's Nathan Marz (creator of Storm), it ended up being quite successful because it was, in fact, a fantastic idea for the time; streaming engines were a bit of a letdown in the correctness department, and batch engines were as inherently unwieldy as you'd expect, so Lambda gave you a way to have your proverbial cake and eat it too. Unfortunately, maintaining a Lambda system is a hassle: you need to build, provision, and maintain two independent versions of your pipeline and then also somehow merge the results from the two pipelines at the end.

As someone who spent years working on a strongly consistent streaming engine, I also found the entire principle of the Lambda Architecture a bit unsavory. Unsurprisingly, I was a huge fan of Jay Kreps' "Questioning the Lambda Architecture" post when it came out. Here was one of the first highly visible statements against the necessity of dual-mode execution. Delightful. Kreps addressed the issue of repeatability in the context of using a replayable system like Kafka as the streaming interconnect, and went so far as to propose the Kappa Architecture, which basically means running a single pipeline using a well-designed system that's appropriately built for the job at hand. I'm not convinced that notion requires its own Greek letter name, but I fully support the idea in principle.

Quite honestly, I'd take things a step further. I would argue that well-designed streaming systems actually provide a strict superset of batch functionality. Modulo perhaps an efficiency delta, there should be no need for batch systems as they exist today. And kudos to the Apache Flink folks for taking this idea to heart and building a system that's all-streaming-all-the-time under the covers, even in "batch" mode; I love it.

Batch and Streaming Efficiency Differences

One which I propose is not an inherent limitation of streaming systems, but simply a consequence of design choices made in most streaming systems thus far. The efficiency delta between batch and streaming is largely the result of the increased bundling and more efficient shuffle transports found in batch systems. Modern batch systems go to great lengths to implement sophisticated optimizations that allow for remarkable levels of throughput using surprisingly modest compute resources. There's no reason the types of clever insights that make batch systems the efficiency heavyweights they are today couldn't be incorporated into a system designed for unbounded data, providing users flexible choice between what we typically consider to be high-latency, higher-efficiency "batch" processing and low-latency, lower-efficiency "streaming" processing. This is effectively what we've done at Google with Cloud Dataflow by providing both batch and streaming runners under the same unified model. In our case, we use separate runners because we happen to have two independently designed systems optimized for their specific use cases. Long term, from an engineering perspective, I'd love to see us merge the two into a single system that incorporates the best parts of both while still maintaining the flexibility of choosing an appropriate efficiency level. But that's not what we have today. And honestly, thanks to the unified Dataflow Model, it's not even strictly necessary; so it may well never happen.

The corollary of all this is that broad maturation of streaming systems combined with robust frameworks for unbounded data processing will in time allow for the relegation of the Lambda Architecture to the antiquity of big data history where it belongs. I believe the time has come to make this a reality. Because to do so—that is, to beat batch at its own game—you really only need two things:

Correctness
: This gets you parity with batch. At the core, correctness boils down to consistent storage. Streaming systems need a method for checkpointing persistent state over time (something Kreps has talked about in his "Why local state is a fundamental primitive in stream processing" post), and it must be well designed enough to remain consistent in light of machine failures. When Spark Streaming first appeared in the public big data scene a few years ago, it was a beacon of consistency in an otherwise dark streaming world. Thankfully, things have improved substantially since then, but it is remarkable how many streaming systems still try to get by without strong consistency.

To reiterate—because this point is important: strong consistency is required for exactly-once processing,[3] which is required for correctness, which is a requirement for any system that's going to have a chance at meeting or exceeding the capabilities of batch systems. Unless you just truly don't care about your results, I implore you to shun any streaming system that doesn't provide strongly consistent state. Batch systems don't require you to verify ahead of time if they are capable of producing correct answers; don't waste your time on streaming systems that can't meet that same bar.

If you're curious to learn more about what it takes to get strong consistency in a streaming system, I recommend you check out the MillWheel, Spark Streaming, and Flink snapshotting papers. All three spend a significant amount of time discussing consistency. Reuven will dive into consistency guarantees later in the book, and if you still find yourself craving more, there's a large amount of quality information on this topic in the literature and elsewhere.

Tools for reasoning about time

This gets you beyond batch. Good tools for reasoning about time are essential for dealing with unbounded, unordered data of varying event-time skew. An increasing number of modern datasets exhibit these characteristics, and existing batch systems (as well as many streaming systems) lack the necessary tools to cope with the difficulties they impose (though this is now rapidly changing, even as I write this). We will spend the bulk of this book explaining and focusing on various facets of this point.

To begin with, we get a basic understanding of the important concept of time domains, after which we take a deeper look at what I mean by unbounded, unordered data of varying event-time skew. We then spend the rest of this chapter looking at common approaches to bounded and unbounded data processing, using both batch and streaming systems.

Event Time Versus Processing Time

To speak cogently about unbounded data processing requires a clear understanding of the domains of time involved. Within any data processing system, there are typically two domains of time that we care about:

3 If you're unfamiliar with what I mean when I say *exactly-once*, it's referring to a specific type of consistency guarantee that certain data processing frameworks provide. Consistency guarantees are typically bucketed into three main classes: at-most-once processing, at-least-once processing, and exactly-once processing. Note that the names in use here refer to the effective semantics as observed within the outputs generated by the pipeline, not the actual number of times a pipeline might process (or attempt to process) any given record. For this reason, the term *effectively-once* is sometimes used instead of exactly-once, since it's more representative of the underlying nature of things. Reuven covers these concepts in much more detail.

Event time
: This is the time at which events actually occurred.

Processing time
: This is the time at which events are observed in the system.

Not all use cases care about event times (and if yours doesn't, hooray! your life is easier), but many do. Examples include characterizing user behavior over time, most billing applications, and many types of anomaly detection, to name a few.

In an ideal world, event time and processing time would always be equal, with events being processed immediately as they occur. Reality is not so kind, however, and the skew between event time and processing time is not only nonzero, but often a highly variable function of the characteristics of the underlying input sources, execution engine, and hardware. Things that can affect the level of skew include the following:

- Shared resource limitations, like network congestion, network partitions, or shared CPU in a nondedicated environment
- Software causes such as distributed system logic, contention, and so on
- Features of the data themselves, like key distribution, variance in throughput, or variance in disorder (i.e., a plane full of people taking their phones out of airplane mode after having used them offline for the entire flight)

As a result, if you plot the progress of event time and processing time in any real-world system, you typically end up with something that looks a bit like the red line in Figure 1-1.

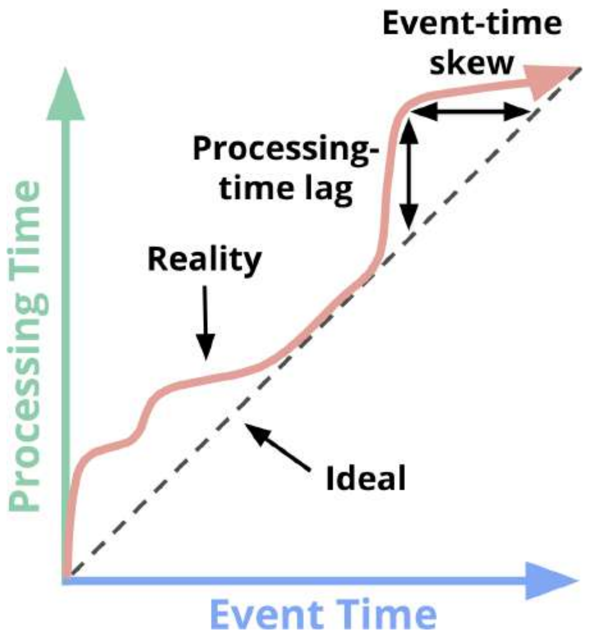


Figure 1-1. Time-domain mapping. The x-axis represents event-time completeness in the system; that is, the time X in event time up to which all data with event times less than X have been observed. The y-axis[4] represents the progress of processing time; that is, normal clock time as observed by the data processing system as it executes.

In Figure 1-1, the black dashed line with slope of 1 represents the ideal, where processing time and event time are exactly equal; the red line represents reality. In this example, the system lags a bit at the beginning of processing time, veers closer toward the ideal in the middle, and then lags again a bit toward the end. At first glance, there are two types of skew visible in this diagram, each in different time domains:

Processing time

The vertical distance between the ideal and the red line is the lag in the processing-time domain. That distance tells you how much delay is observed (in processing time) between when the events for a given time occurred and when

4 Since the original publication of "Streaming 101," numerous individuals have pointed out to me that it would have been more intuitive to place processing time on the x-axis and event time on the y-axis. I do agree that swapping the two axes would initially feel more natural, as event time seems like the dependent variable to processing time's independent variable. However, because both variables are monotonic and intimately related, they're effectively interdependent variables. So I think from a technical perspective you just have to pick an axis and stick with it. Math is confusing (especially outside of North America, where it suddenly becomes plural and gangs up on you).

they were processed. This is the perhaps the more natural and intuitive of the two skews.

Event time
: The horizontal distance between the ideal and the red line is the amount of event-time skew in the pipeline at that moment. It tells you how far behind the ideal (in event time) the pipeline is currently.

In reality, processing-time lag and event-time skew at any given point in time are identical; they're just two ways of looking at the same thing.[5] The important takeaway regarding lag/skew is this: Because the overall mapping between event time and processing time is not static (i.e., the lag/skew can vary arbitrarily over time), this means that you cannot analyze your data solely within the context of when they are observed by your pipeline if you care about their event times (i.e., when the events actually occurred). Unfortunately, this is the way many systems designed for unbounded data have historically operated. To cope with the infinite nature of unbounded datasets, these systems typically provide some notion of windowing the incoming data. We discuss windowing in great depth a bit later, but it essentially means chopping up a dataset into finite pieces along temporal boundaries. If you care about correctness and are interested in analyzing your data in the context of their event times, you cannot define those temporal boundaries using processing time (i.e., processing-time windowing), as many systems do; with no consistent correlation between processing time and event time, some of your event-time data are going to end up in the wrong processing-time windows (due to the inherent lag in distributed systems, the online/offline nature of many types of input sources, etc.), throwing correctness out the window, as it were. We look at this problem in more detail in a number of examples in the sections that follow, as well as the remainder of the book.

Unfortunately, the picture isn't exactly rosy when windowing by event time, either. In the context of unbounded data, disorder and variable skew induce a completeness problem for event-time windows: lacking a predictable mapping between processing time and event time, how can you determine when you've observed all of the data for a given event time *X*? For many real-world data sources, you simply can't. But the vast majority of data processing systems in use today rely on some notion of completeness, which puts them at a severe disadvantage when applied to unbounded datasets.

I propose that instead of attempting to groom unbounded data into finite batches of information that eventually become complete, we should be designing tools that allow us to live in the world of uncertainty imposed by these complex datasets. New data will arrive, old data might be retracted or updated, and any system we build should be able to cope with these facts on its own, with notions of completeness being

5 This result really shouldn't be surprising (but was for me, hence why I'm pointing it out), because we're effectively creating a right triangle with the ideal line when measuring the two types of skew/lag. Maths are cool.

a convenient optimization for specific and appropriate use cases rather than a semantic necessity across all of them.

Before getting into specifics about what such an approach might look like, let's finish up one more useful piece of background: common data processing patterns.

Data Processing Patterns

At this point, we have enough background established that we can begin looking at the core types of usage patterns common across bounded and unbounded data processing today. We look at both types of processing and, where relevant, within the context of the two main types of engines we care about (batch and streaming, where in this context, I'm essentially lumping microbatch in with streaming because the differences between the two aren't terribly important at this level).

Bounded Data

Processing bounded data is conceptually quite straightforward, and likely familiar to everyone. In Figure 1-2, we start out on the left with a dataset full of entropy. We run it through some data processing engine (typically batch, though a well-designed streaming engine would work just as well), such as MapReduce, and on the right side end up with a new structured dataset with greater inherent value.

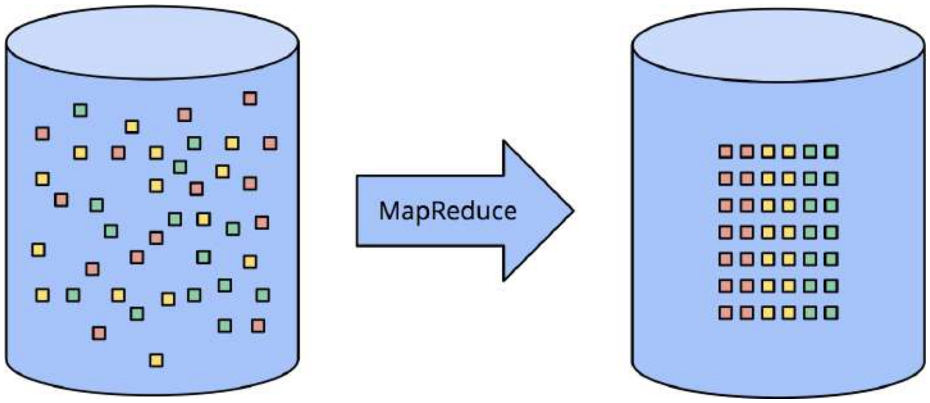


Figure 1-2. Bounded data processing with a classic batch engine. A finite pool of unstructured data on the left is run through a data processing engine, resulting in corresponding structured data on the right.

Though there are of course infinite variations on what you can actually calculate as part of this scheme, the overall model is quite simple. Much more interesting is the task of processing an unbounded dataset. Let's now look at the various ways unbounded data are typically processed, beginning with the approaches used with traditional batch engines and then ending up with the approaches you can take with a system designed for unbounded data, such as most streaming or microbatch engines.

Unbounded Data: Batch

Batch engines, though not explicitly designed with unbounded data in mind, have nevertheless been used to process unbounded datasets since batch systems were first conceived. As you might expect, such approaches revolve around slicing up the unbounded data into a collection of bounded datasets appropriate for batch processing.

Fixed windows

The most common way to process an unbounded dataset using repeated runs of a batch engine is by windowing the input data into fixed-size windows and then processing each of those windows as a separate, bounded data source (sometimes also called *tumbling windows*), as in Figure 1-3. Particularly for input sources like logs, for which events can be written into directory and file hierarchies whose names encode the window they correspond to, this sort of thing appears quite straightforward at first blush because you've essentially performed the time-based shuffle to get data into the appropriate event-time windows ahead of time.

In reality, however, most systems still have a completeness problem to deal with (What if some of your events are delayed en route to the logs due to a network partition? What if your events are collected globally and must be transferred to a common location before processing? What if your events come from mobile devices?), which means some sort of mitigation might be necessary (e.g., delaying processing until you're sure all events have been collected or reprocessing the entire batch for a given window whenever data arrive late).

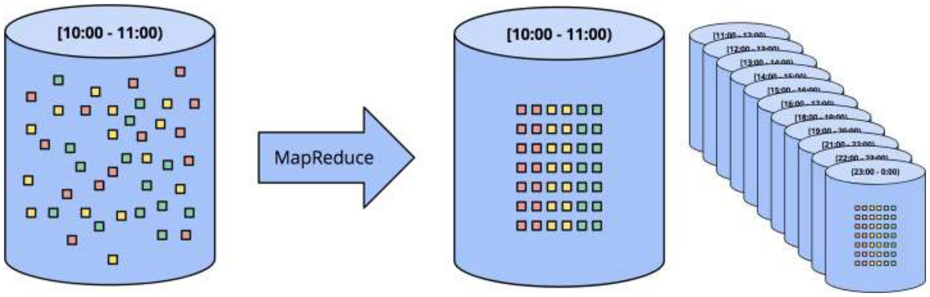


Figure 1-3. Unbounded data processing via ad hoc fixed windows with a classic batch engine. An unbounded dataset is collected up front into finite, fixed-size windows of bounded data that are then processed via successive runs a of classic batch engine.

Sessions

This approach breaks down even more when you try to use a batch engine to process unbounded data into more sophisticated windowing strategies, like sessions. Sessions are typically defined as periods of activity (e.g., for a specific user) terminated by a gap of inactivity. When calculating sessions using a typical batch engine, you often end up with sessions that are split across batches, as indicated by the red marks in Figure 1-4. We can reduce the number of splits by increasing batch sizes, but at the cost of increased latency. Another option is to add additional logic to stitch up sessions from previous runs, but at the cost of further complexity.

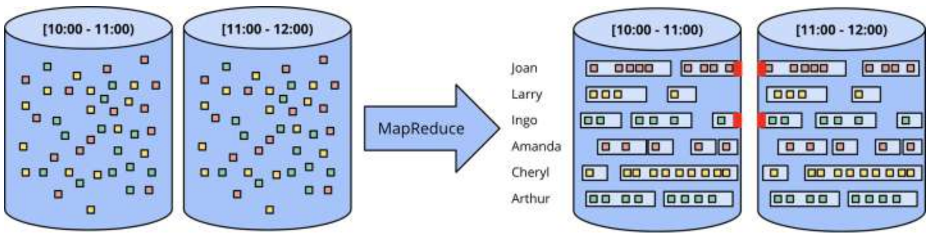


Figure 1-4. Unbounded data processing into sessions via ad hoc fixed windows with a classic batch engine. An unbounded dataset is collected up front into finite, fixed-size windows of bounded data that are then subdivided into dynamic session windows via successive runs a of classic batch engine.

Either way, using a classic batch engine to calculate sessions is less than ideal. A nicer way would be to build up sessions in a streaming manner, which we look at later on.

Unbounded Data: Streaming

Contrary to the ad hoc nature of most batch-based unbounded data processing approaches, streaming systems are built for unbounded data. As we talked about earlier, for many real-world, distributed input sources, you not only find yourself dealing with unbounded data, but also data such as the following:

- Highly unordered with respect to event times, meaning that you need some sort of time-based shuffle in your pipeline if you want to analyze the data in the context in which they occurred.
- Of varying event-time skew, meaning that you can't just assume you'll always see most of the data for a given event time *X* within some constant epsilon of time *Y*.

There are a handful of approaches that you can take when dealing with data that have these characteristics. I generally categorize these approaches into four groups: time-agnostic, approximation, windowing by processing time, and windowing by event time.

Let's now spend a little bit of time looking at each of these approaches.

Time-agnostic

Time-agnostic processing is used for cases in which time is essentially irrelevant; that is, all relevant logic is data driven. Because everything about such use cases is dictated by the arrival of more data, there's really nothing special a streaming engine has to support other than basic data delivery. As a result, essentially all streaming systems in existence support time-agnostic use cases out of the box (modulo system-to-system variances in consistency guarantees, of course, if you care about correctness). Batch systems are also well suited for time-agnostic processing of unbounded data sources by simply chopping the unbounded source into an arbitrary sequence of bounded datasets and processing those datasets independently. We look at a couple of concrete examples in this section, but given the straightforwardness of handling time-agnostic processing (from a temporal perspective at least), we won't spend much more time on it beyond that.

Filtering. A very basic form of time-agnostic processing is filtering, an example of which is rendered in Figure 1-5. Imagine that you're processing web traffic logs and you want to filter out all traffic that didn't originate from a specific domain. You would look at each record as it arrived, see if it belonged to the domain of interest, and drop it if not. Because this sort of thing depends only on a single element at any time, the fact that the data source is unbounded, unordered, and of varying event-time skew is irrelevant.

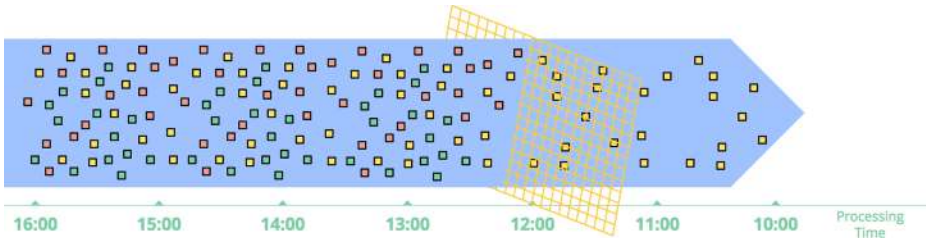


Figure 1-5. Filtering unbounded data. A collection of data (flowing left to right) of varying types is filtered into a homogeneous collection containing a single type.

Inner joins. Another time-agnostic example is an inner join, diagrammed in Figure 1-6. When joining two unbounded data sources, if you care only about the results of a join when an element from both sources arrive, there's no temporal element to the logic. Upon seeing a value from one source, you can simply buffer it up in persistent state; only after the second value from the other source arrives do you need to emit the joined record. (In truth, you'd likely want some sort of garbage collection policy for unemitted partial joins, which would likely be time based. But for a use case with little or no uncompleted joins, such a thing might not be an issue.)

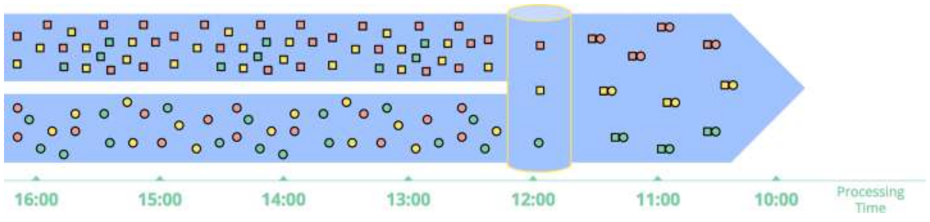


Figure 1-6. Performing an inner join on unbounded data. Joins are produced when matching elements from both sources are observed.

Switching semantics to some sort of outer join introduces the data completeness problem we've talked about: after you've seen one side of the join, how do you know whether the other side is ever going to arrive or not? Truth be told, you don't, so you need to introduce some notion of a timeout, which introduces an element of time. That element of time is essentially a form of windowing, which we'll look at more closely in a moment.

Approximation algorithms

The second major category of approaches is approximation algorithms, such as approximate Top-N, streaming k-means, and so on. They take an unbounded source of input and provide output data that, if you squint at them, look more or less like what you were hoping to get, as in Figure 1-7. The upside of approximation algorithms is that, by design, they are low overhead and designed for unbounded data. The downsides are that a limited set of them exist, the algorithms themselves are often complicated (which makes it difficult to conjure up new ones), and their approximate nature limits their utility.

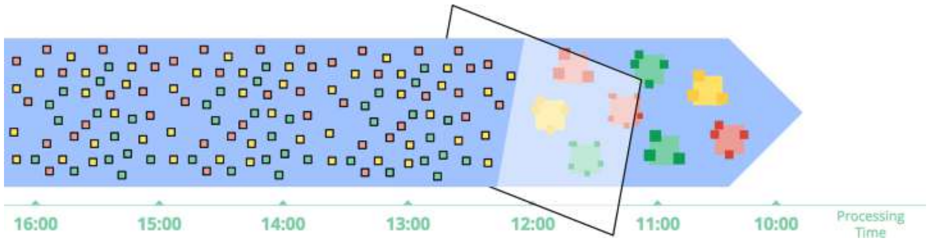


Figure 1-7. Computing approximations on unbounded data. Data are run through a complex algorithm, yielding output data that look more or less like the desired result on the other side.

It's worth noting that these algorithms typically do have some element of time in their design (e.g., some sort of built-in decay). And because they process elements as they arrive, that time element is usually processing-time based. This is particularly important for algorithms that provide some sort of provable error bounds on their approximations. If those error bounds are predicated on data arriving in order, they mean essentially nothing when you feed the algorithm unordered data with varying event-time skew. Something to keep in mind.

Approximation algorithms themselves are a fascinating subject, but as they are essentially another example of time-agnostic processing (modulo the temporal features of the algorithms themselves), they're quite straightforward to use and thus not worth further attention, given our current focus.

Windowing

The remaining two approaches for unbounded data processing are both variations of windowing. Before diving into the differences between them, I should make it clear exactly what I mean by windowing, insomuch as we touched on it only briefly in the previous section. Windowing is simply the notion of taking a data source (either unbounded or bounded), and chopping it up along temporal boundaries into finite chunks for processing. Figure 1-8 shows three different windowing patterns.

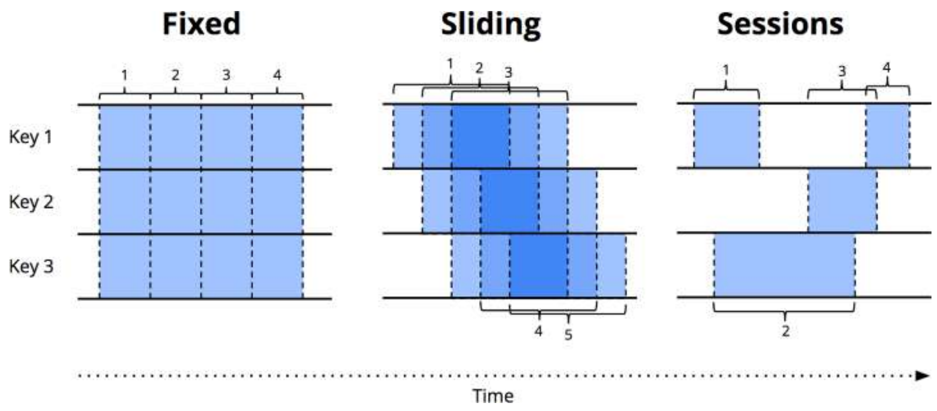


Figure 1-8. Windowing strategies. Each example is shown for three different keys, highlighting the difference between aligned windows (which apply across all the data) and unaligned windows (which apply across a subset of the data).

Let's take a closer look at each strategy:

Fixed windows (aka tumbling windows)
: We discussed fixed windows earlier. Fixed windows slice time into segments with a fixed-size temporal length. Typically (as shown in Figure 1-9), the segments for fixed windows are applied uniformly across the entire dataset, which is an exam-

ple of *aligned* windows. In some cases, it's desirable to phase-shift the windows for different subsets of the data (e.g., per key) to spread window completion load more evenly over time, which instead is an example of *unaligned* windows because they vary across the data.[6]

Sliding windows (aka hopping windows)

A generalization of fixed windows, sliding windows are defined by a fixed length and a fixed period. If the period is less than the length, the windows overlap. If the period equals the length, you have fixed windows. And if the period is greater than the length, you have a weird sort of sampling window that looks only at subsets of the data over time. As with fixed windows, sliding windows are typically aligned, though they can be unaligned as a performance optimization in certain use cases. Note that the sliding windows in Figure 1-8 are drawn as they are to give a sense of sliding motion; in reality, all five windows would apply across the entire dataset.

Sessions

An example of dynamic windows, sessions are composed of sequences of events terminated by a gap of inactivity greater than some timeout. Sessions are commonly used for analyzing user behavior over time, by grouping together a series of temporally related events (e.g., a sequence of videos viewed in one sitting). Sessions are interesting because their lengths cannot be defined a priori; they are dependent upon the actual data involved. They're also the canonical example of unaligned windows because sessions are practically never identical across different subsets of data (e.g., different users).

The two domains of time we discussed earlier (processing time and event time) are essentially the two we care about.[7] Windowing makes sense in both domains, so let's look at each in detail and see how they differ. Because processing-time windowing has historically been more common, we'll start there.

Windowing by processing time. When windowing by processing time, the system essentially buffers up incoming data into windows until some amount of processing time has passed. For example, in the case of five-minute fixed windows, the system would buffer data for five minutes of processing time, after which it would treat all of the data it had observed in those five minutes as a window and send them downstream for processing.

6 We look at aligned fixed windows in detail in Chapter 2.

7 If you poke around enough in the academic literature or SQL-based streaming systems, you'll also come across a third windowing time domain: *tuple-based windowing* (i.e., windows whose sizes are counted in numbers of elements). However, tuple-based windowing is essentially a form of processing-time windowing in which elements are assigned monotonically increasing timestamps as they arrive at the system. As such, we won't discuss tuple-based windowing in detail any further.

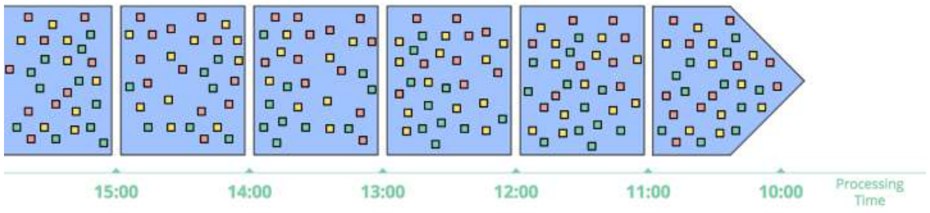


Figure 1-9. Windowing into fixed windows by processing time. Data are collected into windows based on the order they arrive in the pipeline.

There are a few nice properties of processing-time windowing:

- It's simple. The implementation is extremely straightforward because you never worry about shuffling data within time. You just buffer things as they arrive and send them downstream when the window closes.
- Judging window completeness is straightforward. Because the system has perfect knowledge of whether all inputs for a window have been seen, it can make perfect decisions about whether a given window is complete. This means there is no need to be able to deal with "late" data in any way when windowing by processing time.
- If you're wanting to infer information about the source *as it is observed*, processing-time windowing is exactly what you want. Many monitoring scenarios fall into this category. Imagine tracking the number of requests per second sent to a global-scale web service. Calculating a rate of these requests for the purpose of detecting outages is a perfect use of processing-time windowing.

Good points aside, there is one very big downside to processing-time windowing: *if the data in question have event times associated with them, those data must arrive in event-time order if the processing-time windows are to reflect the reality of when those events actually happened.* Unfortunately, event-time ordered data are uncommon in many real-world, distributed input sources.

As a simple example, imagine any mobile app that gathers usage statistics for later processing. For cases in which a given mobile device goes offline for any amount of time (brief loss of connectivity, airplane mode while flying across the country, etc.), the data recorded during that period won't be uploaded until the device comes online again. This means that data might arrive with an event-time skew of minutes, hours, days, weeks, or more. It's essentially impossible to draw any sort of useful inferences from such a dataset when windowed by processing time.

As another example, many distributed input sources might *seem* to provide event-time ordered (or very nearly so) data when the overall system is healthy. Unfortunately, the fact that event-time skew is low for the input source when healthy does not mean it will always stay that way. Consider a global service that processes data collected on multiple continents. If network issues across a bandwidth-constrained transcontinental line (which, sadly, are surprisingly common) further decrease bandwidth and/or increase latency, suddenly a portion of your input data might begin arriving with much greater skew than before. If you are windowing those data by processing time, your windows are no longer representative of the data that actually occurred within them; instead, they represent the windows of time as the events arrived at the processing pipeline, which is some arbitrary mix of old and current data.

What we really want in both of those cases is to window data by their event times in a way that is robust to the order of arrival of events. What we really want is event-time windowing.

Windowing by event time. Event-time windowing is what you use when you need to observe a data source in finite chunks that reflect the times at which those events actually happened. It's the gold standard of windowing. Prior to 2016, most data processing systems in use lacked native support for it (though any system with a decent consistency model, like Hadoop or Spark Streaming 1.x, could act as a reasonable substrate for building such a windowing system). I'm happy to say that the world of today looks very different, with multiple systems, from Flink to Spark to Storm to Apex, natively supporting event-time windowing of some sort.

Figure 1-10 shows an example of windowing an unbounded source into one-hour fixed windows.

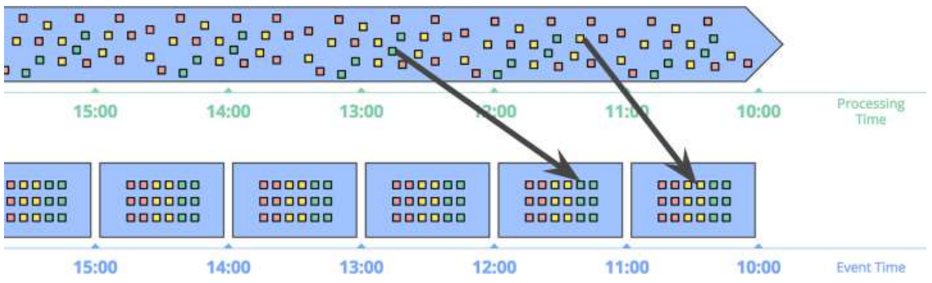


Figure 1-10. Windowing into fixed windows by event time. Data are collected into windows based on the times at which they occurred. The black arrows call out example data that arrived in processing-time windows that differed from the event-time windows to which they belonged.

The black arrows in Figure 1-10 call out two particularly interesting pieces of data. Each arrived in processing-time windows that did not match the event-time windows to which each bit of data belonged. As such, if these data had been windowed into processing-time windows for a use case that cared about event times, the calculated results would have been incorrect. As you would expect, event-time correctness is one nice thing about using event-time windows.

Another nice thing about event-time windowing over an unbounded data source is that you can create dynamically sized windows, such as sessions, without the arbitrary splits observed when generating sessions over fixed windows (as we saw previously in the sessions example from "Unbounded Data: Streaming" on page 18), as demonstrated in Figure 1-11.

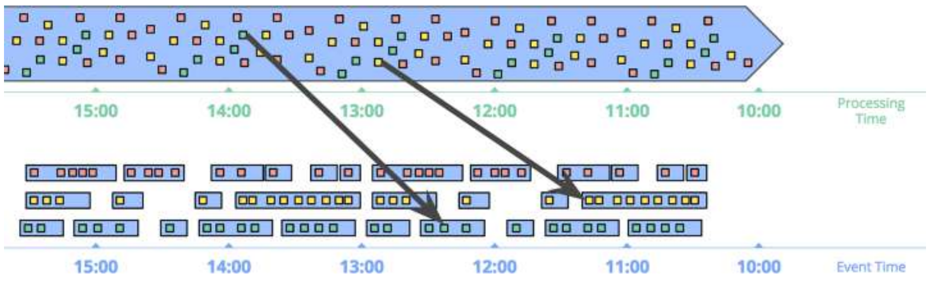


Figure 1-11. Windowing into session windows by event time. Data are collected into session windows capturing bursts of activity based on the times that the corresponding events occurred. The black arrows again call out the temporal shuffle necessary to put the data into their correct event-time locations.

Of course, powerful semantics rarely come for free, and event-time windows are no exception. Event-time windows have two notable drawbacks due to the fact that windows must often live longer (in processing time) than the actual length of the window itself:

Buffering
: Due to extended window lifetimes, more buffering of data is required. Thankfully, persistent storage is generally the cheapest of the resource types most data processing systems depend on (the others being primarily CPU, network bandwidth, and RAM). As such, this problem is typically much less of a concern than you might think when using any well-designed data processing system with strongly consistent persistent state and a decent in-memory caching layer. Also, many useful aggregations do not require the entire input set to be buffered (e.g., sum or average), but instead can be performed incrementally, with a much smaller, intermediate aggregate stored in persistent state.

Completeness
: Given that we often have no good way of knowing when we've seen all of the data for a given window, how do we know when the results for the window are ready

to materialize? In truth, we simply don't. For many types of inputs, the system can give a reasonably accurate heuristic estimate of window completion via something like the watermarks found in MillWheel, Cloud Dataflow, and Flink. But for cases in which absolute correctness is paramount (again, think billing), the only real option is to provide a way for the pipeline builder to express when they want results for windows to be materialized and how those results should be refined over time. Dealing with window completeness (or lack thereof) is a fascinating topic but one perhaps best explored in the context of concrete examples, which we look at next.

Summary

Whew! That was a lot of information. If you've made it this far, you are to be commended! But we are only just getting started. Before forging ahead to looking in detail at the Beam Model approach, let's briefly step back and recap what we've learned so far. In this chapter, we've done the following:

- Clarified terminology, focusing the definition of "streaming" to refer to systems built with unbounded data in mind, while using more descriptive terms like approximate/speculative results for distinct concepts often categorized under the "streaming" umbrella. Additionally, we highlighted two important dimensions of large-scale datasets: cardinality (i.e., bounded versus unbounded) and encoding (i.e., table versus stream), the latter of which will consume much of the second half of the book.
- Assessed the relative capabilities of well-designed batch and streaming systems, positing streaming is in fact a strict superset of batch, and that notions like the Lambda Architecture, which are predicated on streaming being inferior to batch, are destined for retirement as streaming systems mature.
- Proposed two high-level concepts necessary for streaming systems to both catch up to and ultimately surpass batch, those being correctness and tools for reasoning about time, respectively.
- Established the important differences between event time and processing time, characterized the difficulties those differences impose when analyzing data in the context of when they occurred, and proposed a shift in approach away from notions of completeness and toward simply adapting to changes in data over time.
- Looked at the major data processing approaches in common use today for bounded and unbounded data, via both batch and streaming engines, roughly categorizing the unbounded approaches into: time-agnostic, approximation, windowing by processing time, and windowing by event time.

Next up, we dive into the details of the Beam Model, taking a conceptual look at how we've broken up the notion of data processing across four related axes: what, where, when, and how. We also take a detailed look at processing a simple, concrete example dataset across multiple scenarios, highlighting the plurality of use cases enabled by the Beam Model, with some concrete APIs to ground us in reality. These examples will help drive home the notions of event time and processing time introduced in this chapter while additionally exploring new concepts such as watermarks.

CHAPTER 2

The *What*, *Where*, *When*, and *How* of Data Processing

Okay party people, it's time to get concrete!

Chapter 1 focused on three main areas: *terminology*, defining precisely what I mean when I use overloaded terms like "streaming"; *batch versus streaming*, comparing the theoretical capabilities of the two types of systems, and postulating that only two things are necessary to take streaming systems beyond their batch counterparts: correctness and tools for reasoning about time; and *data processing patterns*, looking at the conceptual approaches taken with both batch and streaming systems when processing bounded and unbounded data.

In this chapter, we're now going to focus further on the data processing patterns from Chapter 1, but in more detail, and within the context of concrete examples. By the time we're finished, we'll have covered what I consider to be the core set of principles and concepts required for robust out-of-order data processing; these are the tools for reasoning about time that truly get you beyond classic batch processing.

To give you a sense of what things look like in action, I use snippets of Apache Beam code, coupled with time-lapse diagrams[1] to provide a visual representation of the concepts. Apache Beam is a unified programming model and portability layer for batch and stream processing, with a set of concrete SDKs in various languages (e.g., Java and Python). Pipelines written with Apache Beam can then be portably run on

1 If you're fortunate enough to be reading the Safari version of the book, you have full-blown time-lapse animations just like in "Streaming 102". For print, Kindle, and other ebook versions, there are static images with a link to animated versions on the web.

any of the supported execution engines (Apache Apex, Apache Flink, Apache Spark, Cloud Dataflow, etc.).

I use Apache Beam here for examples not because this is a Beam book (it's not), but because it most completely embodies the concepts described in this book. Back when "Streaming 102" was originally written (back when it was still the Dataflow Model from Google Cloud Dataflow and not the Beam Model from Apache Beam), it was literally the only system in existence that provided the amount of expressiveness necessary for all the examples we'll cover here. A year and a half later, I'm happy to say much has changed, and most of the major systems out there have moved or are moving toward supporting a model that looks a lot like the one described in this book. So rest assured that the concepts we cover here, though informed through the Beam lens, as it were, will apply equally across most other systems you'll come across.

Roadmap

To help set the stage for this chapter, I want to lay out the five main concepts that will underpin all of the discussions therein, and really, for most of the rest of Part 1. We've already covered two of them.

In Chapter 1, I first established the critical distinction between event time (the time that events happen) and processing time (the time they are observed during processing). This provides the foundation for one of the main theses put forth in this book: if you care about both correctness and the context within which events actually occurred, you must analyze data relative to their inherent event times, not the processing time at which they are encountered during the analysis itself.

I then introduced the concept of *windowing* (i.e., partitioning a dataset along temporal boundaries), which is a common approach used to cope with the fact that unbounded data sources technically might never end. Some simpler examples of windowing strategies are *fixed* and *sliding* windows, but more sophisticated types of windowing, such as *sessions* (in which the windows are defined by features of the data themselves; for example, capturing a session of activity per user followed by a gap of inactivity) also see broad usage.

In addition to these two concepts, we're now going to look closely at three more:

Triggers
: A trigger is a mechanism for declaring when the output for a window should be materialized relative to some external signal. Triggers provide flexibility in choosing when outputs should be emitted. In some sense, you can think of them as a flow control mechanism for dictating when results should be materialized. Another way of looking at it is that triggers are like the shutter-release on a camera, allowing you to declare when to take a snapshots in time of the results being computed.

Triggers also make it possible to observe the output for a window multiple times as it evolves. This in turn opens up the door to refining results over time, which allows for providing speculative results as data arrive, as well as dealing with changes in upstream data (revisions) over time or data that arrive late (e.g., mobile scenarios, in which someone's phone records various actions and their event times while the person is offline and then proceeds to upload those events for processing upon regaining connectivity).

Watermarks
A watermark is a notion of input completeness with respect to event times. A watermark with value of time *X* makes the statement: "all input data with event times less than *X* have been observed." As such, watermarks act as a metric of progress when observing an unbounded data source with no known end. We touch upon the basics of watermarks in this chapter, and then Slava goes super deep on the subject in later in the book.

Accumulation
An accumulation mode specifies the relationship between multiple results that are observed for the same window. Those results might be completely disjointed; that is, representing independent deltas over time, or there might be overlap between them. Different accumulation modes have different semantics and costs associated with them and thus find applicability across a variety of use cases.

Also, because I think it makes it easier to understand the relationships between all of these concepts, we revisit the old and explore the new within the structure of answering four questions, all of which I propose are critical to every unbounded data processing problem:

- *What* results are calculated? This question is answered by the types of transformations within the pipeline. This includes things like computing sums, building histograms, training machine learning models, and so on. It's also essentially the question answered by classic batch processing
- *Where* in event time are results calculated? This question is answered by the use of event-time windowing within the pipeline. This includes the common examples of windowing from Chapter 1 (fixed, sliding, and sessions); use cases that seem to have no notion of windowing (e.g., time-agnostic processing; classic batch processing also generally falls into this category); and other, more complex types of windowing, such as time-limited auctions. Also note that it can include processing-time windowing, as well, if you assign ingress times as event times for records as they arrive at the system.
- *When* in processing time are results materialized? This question is answered by the use of triggers and (optionally) watermarks. There are infinite variations on this theme, but the most common patterns are those involving repeated updates

(i.e., materialized view semantics), those that utilize a watermark to provide a single output per window only after the corresponding input is believed to be complete (i.e., classic batch processing semantics applied on a per-window basis), or some combination of the two.

- *How* do refinements of results relate? This question is answered by the type of accumulation used: discarding (in which results are all independent and distinct), accumulating (in which later results build upon prior ones), or accumulating and retracting (in which both the accumulating value plus a retraction for the previously triggered value(s) are emitted).

We look at each of these questions in much more detail throughout the rest of the book. And, yes, I'm going to run this color scheme thing into the ground in an attempt to make it abundantly clear which concepts relate to which question in the *What*/*Where*/*When*/*How* idiom. You're welcome <winky-smiley/>.[2]

Batch Foundations: *What* and *Where*

Okay, let's get this party started. First stop: batch processing.

What: Transformations

The transformations applied in classic batch processing answer the question: "*What* results are calculated?" Even though you are likely already familiar with classic batch processing, we're going to start there anyway because it's the foundation on top of which we add all of the other concepts.

In the rest of this chapter (and indeed, through much of the book), we look at a single example: computing keyed integer sums over a simple dataset consisting of nine values. Let's imagine that we've written a team-based mobile game and we want to build a pipeline that calculates team scores by summing up the individual scores reported by users' phones. If we were to capture our nine example scores in a SQL table named "UserScores," it might look something like this:

```
> SELECT * FROM UserScores ORDER BY EventTime;
------------------------------------------------
| Name  | Team  | Score | EventTime | ProcTime |
------------------------------------------------
| Julie | TeamX |     5 |  12:00:26 | 12:05:19 |
| Frank | TeamX |     9 |  12:01:26 | 12:08:19 |
| Ed    | TeamX |     7 |  12:02:26 | 12:05:39 |
| Julie | TeamX |     8 |  12:03:06 | 12:07:06 |
| Amy   | TeamX |     3 |  12:03:39 | 12:06:13 |
```

2 Bear with me here. Fine-grained emotional expressions via composite punctuation (i.e., emoticons) are strictly forbidden in O'Reilly publications <winky-smiley/>.

```
| Fred  | TeamX |     4 |  12:04:19 | 12:06:39 |
| Naomi | TeamX |     3 |  12:06:39 | 12:07:19 |
| Becky | TeamX |     8 |  12:07:26 | 12:08:39 |
| Naomi | TeamX |     1 |  12:07:46 | 12:09:00 |
------------------------------------------------
```

Note that all the scores in this example are from users on the same team; this is to keep the example simple, given that we have a limited number of dimensions in our diagrams that follow. And because we're grouping by team, we really just care about the last three columns:

`Score`
: The individual user score associated with this event

`EventTime`
: The event time for the score; that is, the time at which the score occurred

`ProcTime`
: The processing for the score; that is, the time at which the score was observed by the pipeline

For each example pipeline, we'll look at a time-lapse diagram that highlights how the data evolves over time. Those diagrams plot our nine scores in the two dimensions of time we care about: event time in the x-axis, and processing time in the y-axis. Figure 2-1 illustrates what a static plot of the input data looks like.

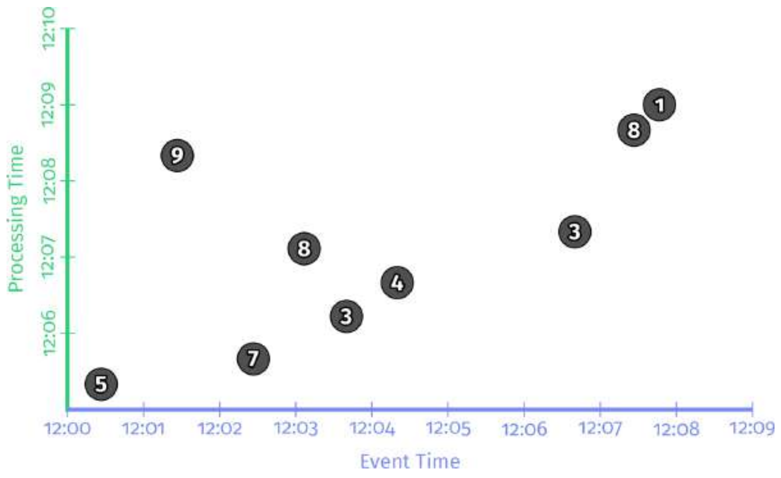


Figure 2-1. Nine input records, plotted in both event time and processing time

Subsequent time-lapse diagrams are either animations (Safari) or a sequence of frames (print and all other digital formats), allowing you to see how the data are processed over time (more on this shortly after we get to the first time-lapse diagram).

Preceding each example is a short snippet of Apache Beam Java SDK pseudocode to make the definition of the pipeline more concrete. It is pseudocode in the sense that I sometime bend the rules to make the examples clearer, elide details (like the use of concrete I/O sources), or simplify names (the trigger names in Beam Java 2.x and earlier are painfully verbose; I use simpler names for clarity). Beyond minor things like those, it's otherwise real-world Beam code (and real code is available on GitHub for all examples in this chapter).

If you're already familiar with something like Spark or Flink, you should have a relatively easy time understanding what the Beam code is doing. But to give you a crash course in things, there are two basic primitives in Beam:

`PCollections`
: These represent datasets (possibly massive ones) across which parallel transformations can be performed (hence the "P" at the beginning of the name).

`PTransforms`
: These are applied to `PCollections` to create new `PCollections`. `PTransforms` may perform element-wise transformations, they may group/aggregate multiple elements together, or they may be a composite combination of other `PTransforms`, as depicted in Figure 2-2.

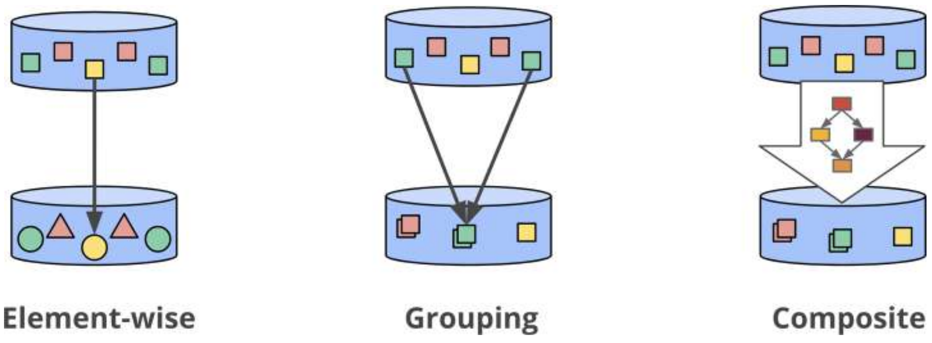


Figure 2-2. Types of transformations

For the purposes of our examples, we typically assume that we start out with a preloaded `PCollection<KV<Team, Integer>>` named "input" (that is, a `PCollection` composed of key/value pairs of `Teams` and `Integers`, where the `Teams` are just something like `Strings` representing team names, and the `Integers` are scores from any individual on the corresponding team). In a real-world pipeline, we would've acquired input by reading in a `PCollection<String>` of raw data (e.g., log records) from an I/O source and then transforming it into a `PCollection<KV<Team, Integer>>` by parsing the log records into appropriate key/value pairs. For the sake of clarity in this first example, I include pseudocode for all of those steps, but in subsequent examples, I elide the I/O and parsing.

Thus, for a pipeline that simply reads in data from an I/O source, parses team/score pairs, and calculates per-team sums of scores, we'd have something like that shown in Example 2-1.

Example 2-1. Summation pipeline

```
PCollection<String> raw = IO.read(...);
PCollection<KV<Team, Integer>> input = raw.apply(new ParseFn());
PCollection<KV<Team, Integer>> totals =
  input.apply(Sum.integersPerKey());
```

Key/value data are read from an I/O source, with a `Team` (e.g., `String` of the team name) as the key and an `Integer` (e.g., individual team member scores) as the value. The values for each key are then summed together to generate per-key sums (e.g., total team score) in the output collection.

For all the examples to come, after seeing a code snippet describing the pipeline that we're analyzing, we'll then look at a time-lapse diagram showing the execution of that pipeline over our concrete dataset for a single key. In a real pipeline, you can imagine that similar operations would be happening in parallel across multiple machines, but for the sake of our examples, it will be clearer to keep things simple.

As noted previously, Safari editions present the complete execution as an animated movie, whereas print and all other digital formats use a static sequence of key frames that provide a sense of how the pipeline progresses over time. In both cases, we also provide a URL to a fully animated version on *www.streamingbook.net*.

Each diagram plots the inputs and outputs across two dimensions: event time (on the x-axis) and processing time (on the y-axis). Thus, real time as observed by the pipeline progresses from bottom to top, as indicated by the thick horizontal black line that ascends in the processing-time axis as time progresses. Inputs are circles, with the number inside the circle representing the value of that specific record. They start out light gray, and darken as the pipeline observes them.

As the pipeline observes values, it accumulates them in its intermediate state and eventually materializes the aggregate results as output. State and output are represented by rectangles (gray for state, blue for output), with the aggregate value near the top, and with the area covered by the rectangle representing the portions of event time and processing time accumulated into the result. For the pipeline in Example 2-1, it would look something like that shown in Figure 2-3 when executed on a classic batch engine.

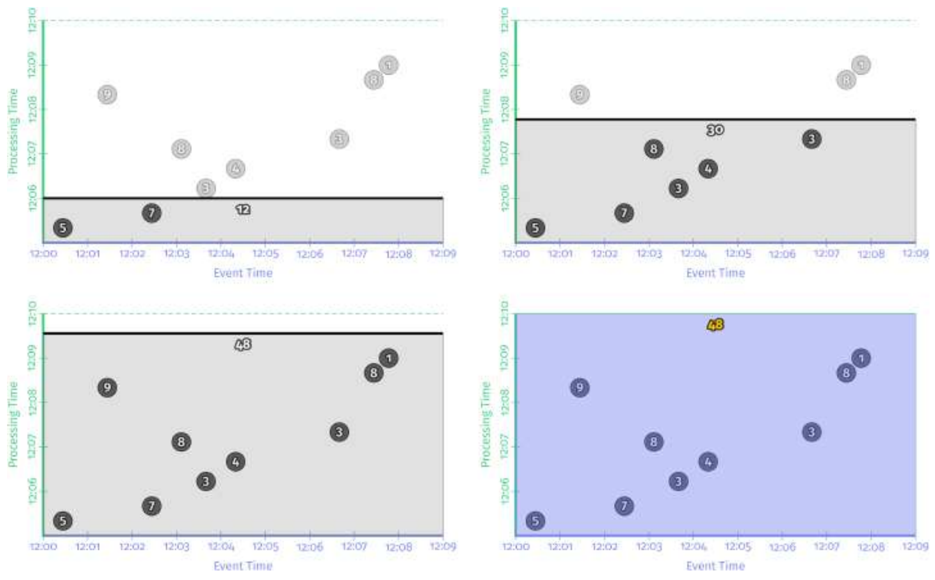


Figure 2-3. Classic batch processing

Because this is a batch pipeline, it accumulates state until it's seen all of the inputs (represented by the dashed green line at the top), at which point it produces its single output of 48. In this example, we're calculating a sum over all of event time because we haven't applied any specific windowing transformations; hence the rectangles for state and output cover the entirety of the x-axis. If we want to process an unbounded data source, however, classic batch processing won't be sufficient; we can't wait for the input to end, because it effectively never will. One of the concepts we want is windowing, which we introduced in Chapter 1. Thus, within the context of our second question—"*Where* in event time are results calculated?"—we'll now briefly revisit windowing.

Where: Windowing

As discussed in Chapter 1, windowing is the process of slicing up a data source along temporal boundaries. Common windowing strategies include fixed windows, sliding windows, and sessions windows, as demonstrated in Figure 2-4.

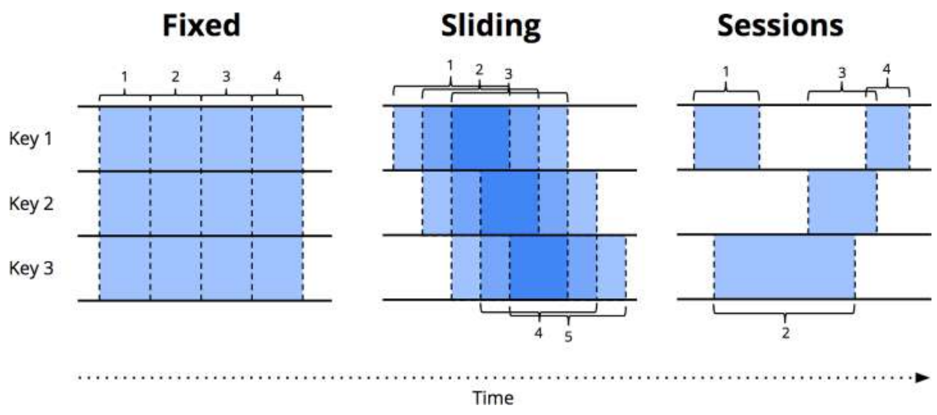


Figure 2-4. Example windowing strategies. Each example is shown for three different keys, highlighting the difference between aligned windows (which apply across all the data) and unaligned windows (which apply across a subset of the data).

To get a better sense of what windowing looks like in practice, let's take our integer summation pipeline and window it into fixed, two-minute windows. With Beam, the change is a simple addition of a `Window.into` transform, which you can see highlighted in Example 2-2.

Example 2-2. Windowed summation code

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES)))
  .apply(Sum.integersPerKey());
```

Recall that Beam provides a unified model that works in both batch and streaming because semantically batch is really just a subset of streaming. As such, let's first execute this pipeline on a batch engine; the mechanics are more straightforward, and it will give us something to directly compare against when we switch to a streaming engine. Figure 2-5 presents the result.

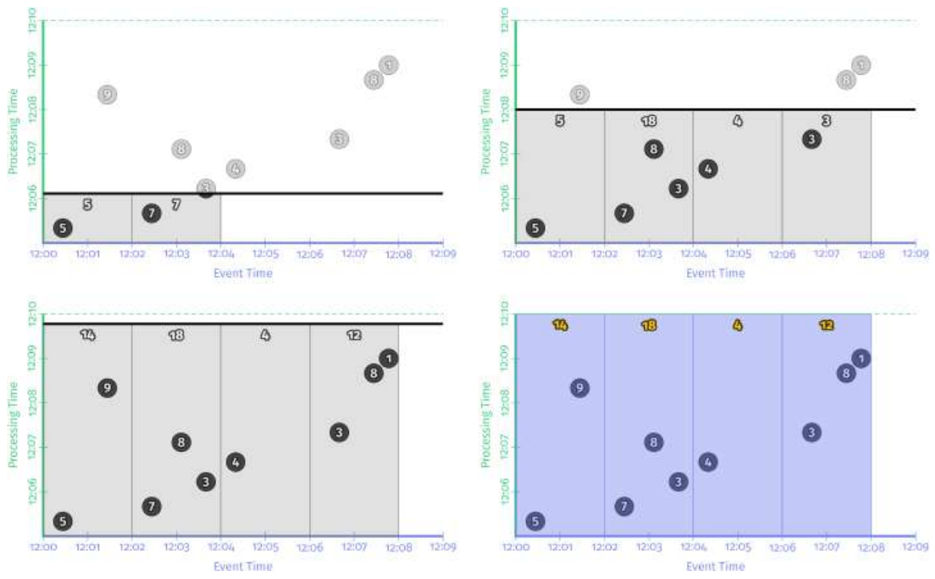


Figure 2-5. Windowed summation on a batch engine

As before, inputs are accumulated in state until they are entirely consumed, after which output is produced. In this case, however, instead of one output, we get four: a single output, for each of the four relevant two-minute event-time windows.

At this point we've revisited the two main concepts that I introduced in Chapter 1: the relationship between the event-time and processing-time domains, and windowing. If we want to go any further, we'll need to start adding the new concepts mentioned at the beginning of this section: triggers, watermarks, and accumulation.

Going Streaming: *When* and *How*

We just observed the execution of a windowed pipeline on a batch engine. But, ideally, we'd like to have lower latency for our results, and we'd also like to natively handle unbounded data sources. Switching to a streaming engine is a step in the right direction, but our previous strategy of waiting until our input has been consumed in its entirety to generate output is no longer feasible. Enter triggers and watermarks.

When: The Wonderful Thing About Triggers Is Triggers Are Wonderful Things!

Triggers provide the answer to the question: "*When* in processing time are results materialized?" Triggers declare when output for a window should happen in processing time (though the triggers themselves might make those decisions based on things that happen in other time domains, such as watermarks progressing in the event-time domain, as we'll see in a few moments). Each specific output for a window is referred to as a *pane* of the window.

Though it's possible to imagine quite a breadth of possible triggering semantics,[3] conceptually there are only two generally useful types of triggers, and practical applications almost always boil down using either one or a combination of both:

Repeated update triggers
: These periodically generate updated panes for a window as its contents evolve. These updates can be materialized with every new record, or they can happen after some processing-time delay, such as once a minute. The choice of period for a repeated update trigger is primarily an exercise in balancing latency and cost.

Completeness triggers
: These materialize a pane for a window only after the input for that window is believed to be complete to some threshold. This type of trigger is most analogous to what we're familiar with in batch processing: only after the input is complete do we provide a result. The difference in the trigger-based approach is that the notion of completeness is scoped to the context of a single window, rather than always being bound to the completeness of the entire input.

Repeated update triggers are the most common type of trigger encountered in streaming systems. They are simple to implement and simple to understand, and they provide useful semantics for a specific type of use case: repeated (and eventually consistent) updates to a materialized dataset, analogous to the semantics you get with materialized views in the database world.

Completeness triggers are less frequently encountered, but provide streaming semantics that more closely align with those from the classic batch processing world. They also provide tools for reasoning about things like missing data and late data, both of which we discuss shortly (and in the next chapter) as we explore the underlying primitive that drives completeness triggers: watermarks.

But first, let's start simple and look at some basic repeated update triggers in action. To make the notion of triggers a bit more concrete, let's go ahead and add the most straightforward type of trigger to our example pipeline: a trigger that fires with every new record, as shown in Example 2-3.

3 And indeed, we did just that with the original triggers feature in Beam. In retrospect, we went a bit overboard. Future iterations will be simpler and easier to use, and in this book I focus only on the pieces that are likely to remain in some form or another.

Example 2-3. Triggering repeatedly with every record

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
                .triggering(Repeatedly(AfterCount(1))));
  .apply(Sum.integersPerKey());
```

If we were to run this new pipeline on a streaming engine, the results would look something like that shown in Figure 2-6.

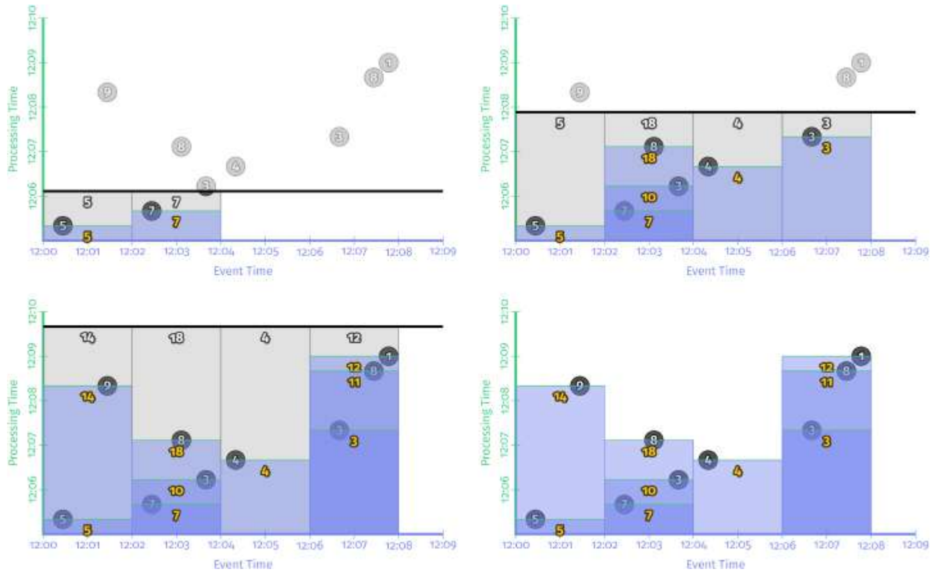


Figure 2-6. Per-record triggering on a streaming engine

You can see how we now get multiple outputs (panes) for each window: once per corresponding input. This sort of triggering pattern works well when the output stream is being written to some sort of table that you can simply poll for results. Any time you look in the table, you'll see the most up-to-date value for a given window, and those values will converge toward correctness over time.

One downside of per-record triggering is that it's quite chatty. When processing large-scale data, aggregations like summation provide a nice opportunity to reduce the cardinality of the stream without losing information. This is particularly noticeable for cases in which you have high-volume keys; for our example, massive teams with lots of active players. Imagine a massively multiplayer game in which players are split into one of two factions, and you want to tally stats on a per-faction basis. It's probably unnecessary to update your tallies with every new input record for every player in a given faction. Instead, you might be happy updating them after some processing-time delay, say every second, or every minute. The nice side effect of using processing-time delays is that it has an equalizing effect across high-volume keys or windows: the resulting stream ends up being more uniform cardinality-wise.

There are two different approaches to processing-time delays in triggers: *aligned delays* (where the delay slices up processing time into fixed regions that align across keys and windows) and *unaligned delays* (where the delay is relative to the data observed within a given window). A pipeline with unaligned delays might look like Example 2-4, the results of which are shown in Figure 2-7.

Example 2-4. Triggering on aligned two-minute processing-time boundaries

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(Repeatedly(AlignedDelay(TWO_MINUTES)))
  .apply(Sum.integersPerKey());
```

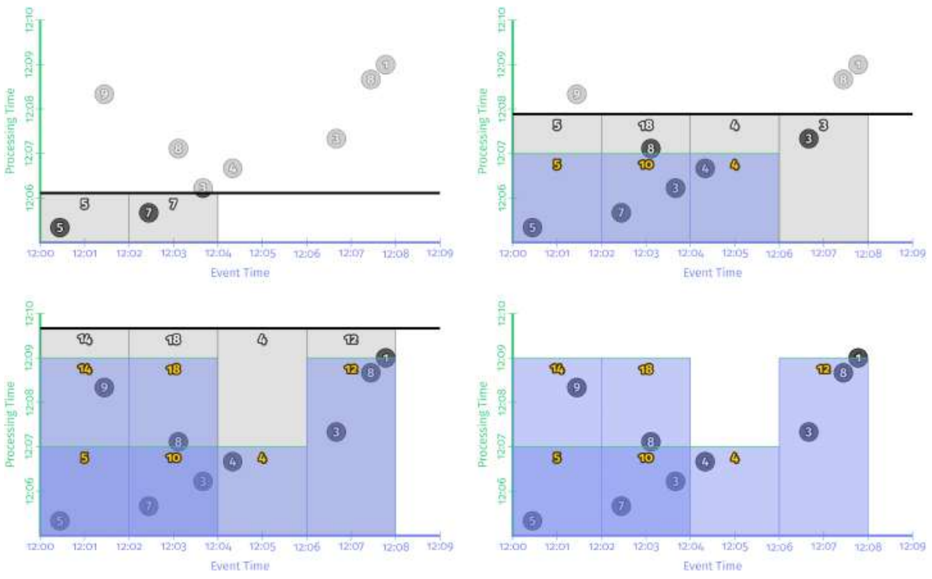


Figure 2-7. Two-minute aligned delay triggers (i.e., microbatching)

This sort of aligned delay trigger is effectively what you get from a microbatch streaming system like Spark Streaming. The nice thing about it is predictability; you get regular updates across all modified windows at the same time. That's also the downside: all updates happen at once, which results in bursty workloads that often require greater peak provisioning to properly handle the load. The alternative is to use an unaligned delay. That would look something Example 2-5 in Beam. Figure 2-8 presents the results.

Example 2-5. Triggering on unaligned two-minute processing-time boundaries

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(Repeatedly(UnalignedDelay(TWO_MINUTES))
  .apply(Sum.integersPerKey());
```

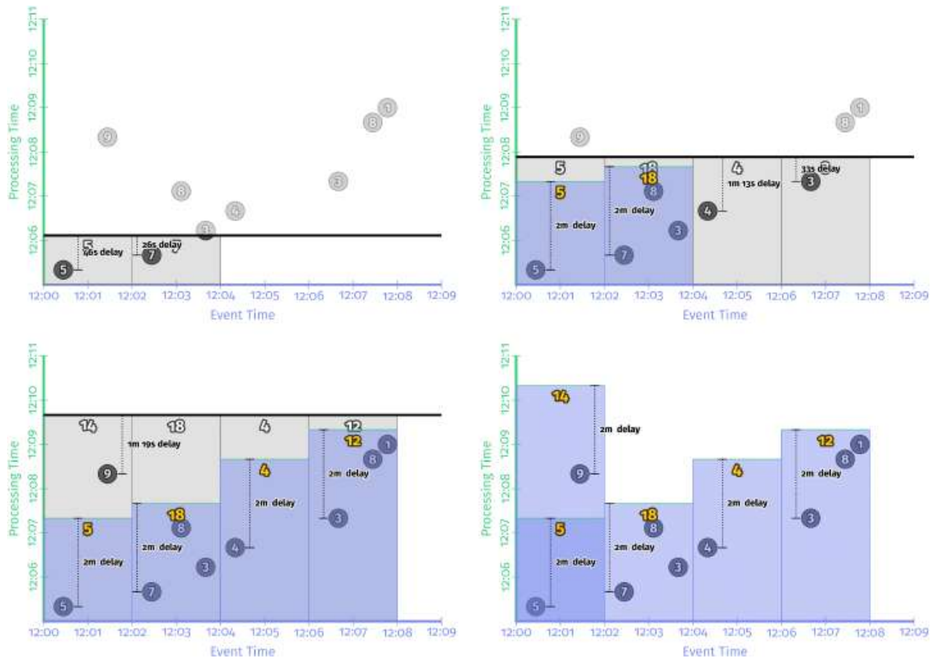


Figure 2-8. Two-minute unaligned delay triggers

Contrasting the unaligned delays in Figure 2-8 to the aligned delays in Figure 2-6, it's easy to see how the unaligned delays spread the load out more evenly across time. The actual latencies involved for any given window differ between the two, sometimes more and sometimes less, but in the end the average latency will remain essentially the same. From that perspective, unaligned delays are typically the better choice for large-scale processing because they result in a more even load distribution over time.

Repeated update triggers are great for use cases in which we simply want periodic updates to our results over time and are fine with those updates converging toward correctness with no clear indication of when correctness is achieved. However, as we discussed in Chapter 1, the vagaries of distributed systems often lead to a varying level of skew between the time an event happens and the time it's actually observed by your pipeline, which means it can be difficult to reason about when your output presents an accurate and complete view of your input data. For cases in which input completeness matters, it's important to have some way of reasoning about completeness rather than blindly trusting the results calculated by whichever subset of data happen to have found their way to your pipeline. Enter watermarks.

When: Watermarks

Watermarks are a supporting aspect of the answer to the question: "*When* in processing time are results materialized?" Watermarks are temporal notions of input completeness in the event-time domain. Worded differently, they are the way the system measures progress and completeness relative to the event times of the records being processed in a stream of events (either bounded or unbounded, though their usefulness is more apparent in the unbounded case).

Recall this diagram from Chapter 1, slightly modified in Figure 2-9, in which I described the skew between event time and processing time as an ever-changing function of time for most real-world distributed data processing systems.

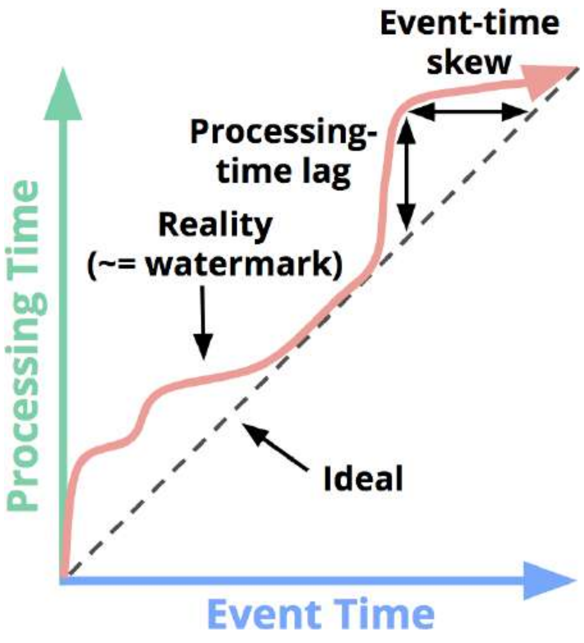


Figure 2-9. Event-time progress, skew, and watermarks

That meandering red line that I claimed represented reality is essentially the watermark; it captures the progress of event-time completeness as processing time progresses. Conceptually, you can think of the watermark as a function, $F(P) \rightarrow E$, which takes a point in processing time and returns a point in event time.[4] That point in event time, *E*, is the point up to which the system believes all inputs with event times less than *E* have been observed. In other words, it's an assertion that no more data with event times less than *E* will ever be seen again. Depending upon the type of watermark, perfect or heuristic, that assertion can be a strict guarantee or an educated guess, respectively:

Perfect watermarks
: For the case in which we have perfect knowledge of all of the input data, it's possible to construct a perfect watermark. In such a case, there is no such thing as late data; all data are early or on time.

Heuristic watermarks
: For many distributed input sources, perfect knowledge of the input data is impractical, in which case the next best option is to provide a heuristic watermark. Heuristic watermarks use whatever information is available about the inputs (partitions, ordering within partitions if any, growth rates of files, etc.) to provide an estimate of progress that is as accurate as possible. In many cases, such watermarks can be remarkably accurate in their predictions. Even so, the use of a heuristic watermark means that it might sometimes be wrong, which will lead to late data. We show you about ways to deal with late data soon.

Because they provide a notion of completeness relative to our inputs, watermarks form the foundation for the second type of trigger mentioned previously: *completeness triggers*. Watermarks themselves are a fascinating and complex topic, as you'll see when you get to Slava's watermarks deep dive. But for now, let's look at them in action by updating our example pipeline to utilize a completeness trigger built upon watermarks, as demonstrated in Example 2-6.

Example 2-6. Watermark completeness trigger

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(AfterWatermark()))
  .apply(Sum.integersPerKey());
```

Now, an interesting quality of watermarks is that they are a class of functions, meaning there are multiple different functions $F(P) \rightarrow E$ that satisfy the properties of a watermark, to varying degrees of success. As I noted earlier, for situations in which you have perfect knowledge of your input data, it might be possible to build a perfect

4 More accurately, the input to the function is really the state at time *P* of everything upstream of the point in the pipeline where the watermark is being observed: the input source, buffered data, data actively being processed, and so on; but conceptually it's simpler to think of it as a mapping from processing time to event time.

watermark, which is the ideal situation. But for cases in which you lack perfect knowledge of the inputs or for which it's simply too computationally expensive to calculate the perfect watermark, you might instead choose to utilize a heuristic for defining your watermark. The point I want to make here is that the given watermark algorithm in use is independent from the pipeline itself. We're not going to discuss in detail what it means to implement a watermark here. For now, to help drive home this idea that a given input set can have different watermarks applied to it, let's take a look at our pipeline in Example 2-6 when executed on the same dataset but using two distinct watermark implementations (Figure 2-10): on the left, a perfect watermark; on the right, a heuristic watermark.

In both cases, windows are materialized as the watermark passes the end of the window. The perfect watermark, as you might expect, perfectly captures the event-time completeness of the pipeline as time progresses. In contrast, the specific algorithm used for the heuristic watermark on the right fails to take the value of 9 into account,[5] which drastically changes the shape of the materialized outputs, both in terms of output latency and correctness (as seen by the incorrect answer of 5 that's provided for the [12:00, 12:02) window).

The big difference between the watermark triggers from Figure 2-9 and the repeated update triggers we saw in Figures 2-5 through 2-7 is that the *watermarks give us a way to reason about the completeness of our input*. Until the system materializes an output for a given window, we know that the system does not yet believe the inputs to be complete. This is especially important for use cases in which you want to reason about a *lack of data* in the input, or *missing data*.

5 Note that I specifically chose to omit the value of 9 from the heuristic watermark because it will help me to make some important points about late data and watermark lag. In reality, a heuristic watermark might be just as likely to omit some other value(s) instead, which in turn could have significantly less drastic effect on the watermark. If winnowing late-arriving data from the watermark is your goal (which is very valid in some cases, such as abuse detection, for which you just want to see a significant majority of the data as quickly as possible), you don't necessarily want a heuristic watermark rather than a perfect watermark. What you really want is a percentile watermark, which explicitly drops some percentile of late-arriving data from its calculations.

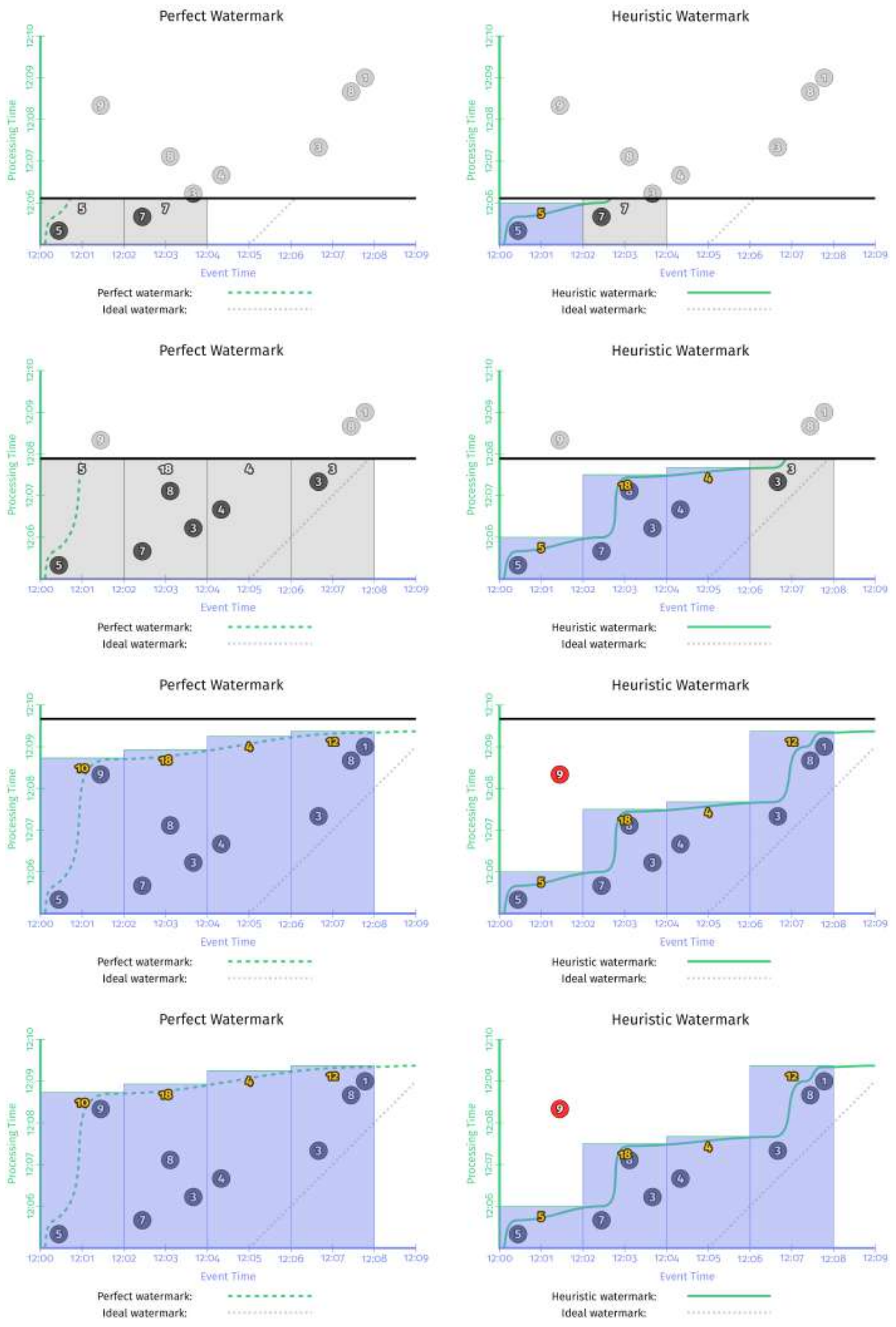


Figure 2-10. Windowed summation on a streaming engine with perfect (left) and heuristic (right) watermarks

A great example of a missing-data use case is outer joins. Without a notion of completeness like watermarks, how do you know when to give up and emit a partial join rather than continue to wait for that join to complete? You don't. And basing that decision on a processing-time delay, which is the common approach in streaming systems that lack true watermark support, is not a safe way to go, because of the variable nature of event-time skew we spoke about in Chapter 1: as long as skew remains smaller than the chosen processing-time delay, your missing-data results will be correct, but any time skew grows beyond that delay, they will suddenly become *in*correct. From this perspective, event-time watermarks are a critical piece of the puzzle for many real-world streaming use cases which must reason about a lack of data in the input, such as outer joins, anomaly detection, and so on.

Now, with that said, these watermark examples also highlight two *shortcomings* of watermarks (and any other notion of completeness), specifically that they can be one of the following:

Too slow

When a watermark of any type is correctly delayed due to known unprocessed data (e.g., slowly growing input logs due to network bandwidth constraints), that translates directly into delays in output if advancement of the watermark is the only thing you depend on for stimulating results.

This is most obvious in the left diagram of Figure 2-10, for which the late arriving 9 holds back the watermark for all the subsequent windows, even though the input data for those windows become complete earlier. This is particularly apparent for the second window, [12:02, 12:04), for which it takes nearly seven minutes from the time the first value in the window occurs until we see any results for the window whatsoever. The heuristic watermark in this example doesn't suffer the same issue quite so badly (five minutes until output), but don't take that to mean heuristic watermarks never suffer from watermark lag; that's really just a consequence of the record I chose to omit from the heuristic watermark in this specific example.

The important point here is the following: Although watermarks provide a very useful notion of completeness, depending upon completeness for producing output is often not ideal from a latency perspective. Imagine a dashboard that contains valuable metrics, windowed by hour or day. It's unlikely you'd want to wait a full hour or day to begin seeing results for the current window; that's one of the pain points of using classic batch systems to power such systems. Instead, it would be much nicer to see the results for those windows refine over time as the inputs evolve and eventually become complete.

Too fast

When a heuristic watermark is incorrectly advanced earlier than it should be, it's possible for data with event times before the watermark to arrive some time later, creating late data. This is what happened in the example on the right: the watermark advanced past the end of the first window before all the input data for that window had been observed, resulting in an incorrect output value of 5 instead of 14. This shortcoming is strictly a problem with heuristic watermarks; their heuristic nature implies they will sometimes be wrong. As a result, relying on them alone for determining when to materialize output is insufficient if you care about correctness.

In Chapter 1, I made some rather emphatic statements about notions of completeness being insufficient for most use cases requiring robust out-of-order processing of unbounded data streams. These two shortcomings—watermarks being too slow or too fast—are the foundations for those arguments. You simply cannot get both low latency and correctness out of a system that relies solely on notions of completeness.[6] So, for cases for which you do want the best of both worlds, what's a person to do? Well, if repeated update triggers provide low-latency updates but no way to reason about completeness, and watermarks provide a notion of completeness but variable and possible high latency, why not combine their powers together?

When: Early/On-Time/Late Triggers FTW!

We've now looked at the two main types of triggers: repeated update triggers and completeness/watermark triggers. In many case, neither of them alone is sufficient, but the combination of them together is. Beam recognizes this fact by providing an extension of the standard watermark trigger that also supports repeated update triggering on either side of the watermark. This is known as the early/on-time/late trigger because it partitions the panes that are materialized by the compound trigger into three categories:

- Zero or more *early panes*, which are the result of a repeated update trigger that periodically fires up until the watermark passes the end of the window. The panes generated by these firings contain speculative results, but allow us to observe the evolution of the window over time as new input data arrive. This compensates for the shortcoming of watermarks sometimes being *too slow*.
- A single *on-time pane*, which is the result of the completeness/watermark trigger firing after the watermark passes the end of the window. This firing is special because it provides an assertion that the system now believes the input for this

6 Which isn't to say there aren't use cases that care primarily about correctness and not so much about latency; in those cases, using an accurate watermark as the sole driver of output from a pipeline is a reasonable approach.

window to be complete.[7] This means that it is now safe to reason about *missing data*; for example, to emit a partial join when performing an outer join.

- Zero or more *late panes*, which are the result of another (possibly different) repeated update trigger that periodically fires any time late data arrive after the watermark has passed the end of the window. In the case of a perfect watermark, there will always be zero late panes. But in the case of a heuristic watermark, any data the watermark failed to properly account for will result in a late firing. This compensates for the shortcoming of watermarks being *too fast*.

Let's see how this looks in action. We'll update our pipeline to use a periodic processing-time trigger with an aligned delay of one minute for the early firings, and a per-record trigger for the late firings. That way, the early firings will give us some amount of batching for high-volume windows (thanks to the fact that the trigger will fire only once per minute, regardless of the throughput into the window), but we won't introduce unnecessary latency for the late firings, which are hopefully somewhat rare if we're using a reasonably accurate heuristic watermark. In Beam, that looks Example 2-7 (Figure 2-11 shows the results).

Example 2-7. Early, on-time, and late firings via the early/on-time/late API

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(AfterWatermark()
                            .withEarlyFirings(AlignedDelay(ONE_MINUTE))
                            .withLateFirings(AfterCount(1))))
  .apply(Sum.integersPerKey());
```

7 And, as we know from before, this assertion is either guaranteed, in the case of a perfect watermark being used, or an educated guess, in the case of a heuristic watermark.

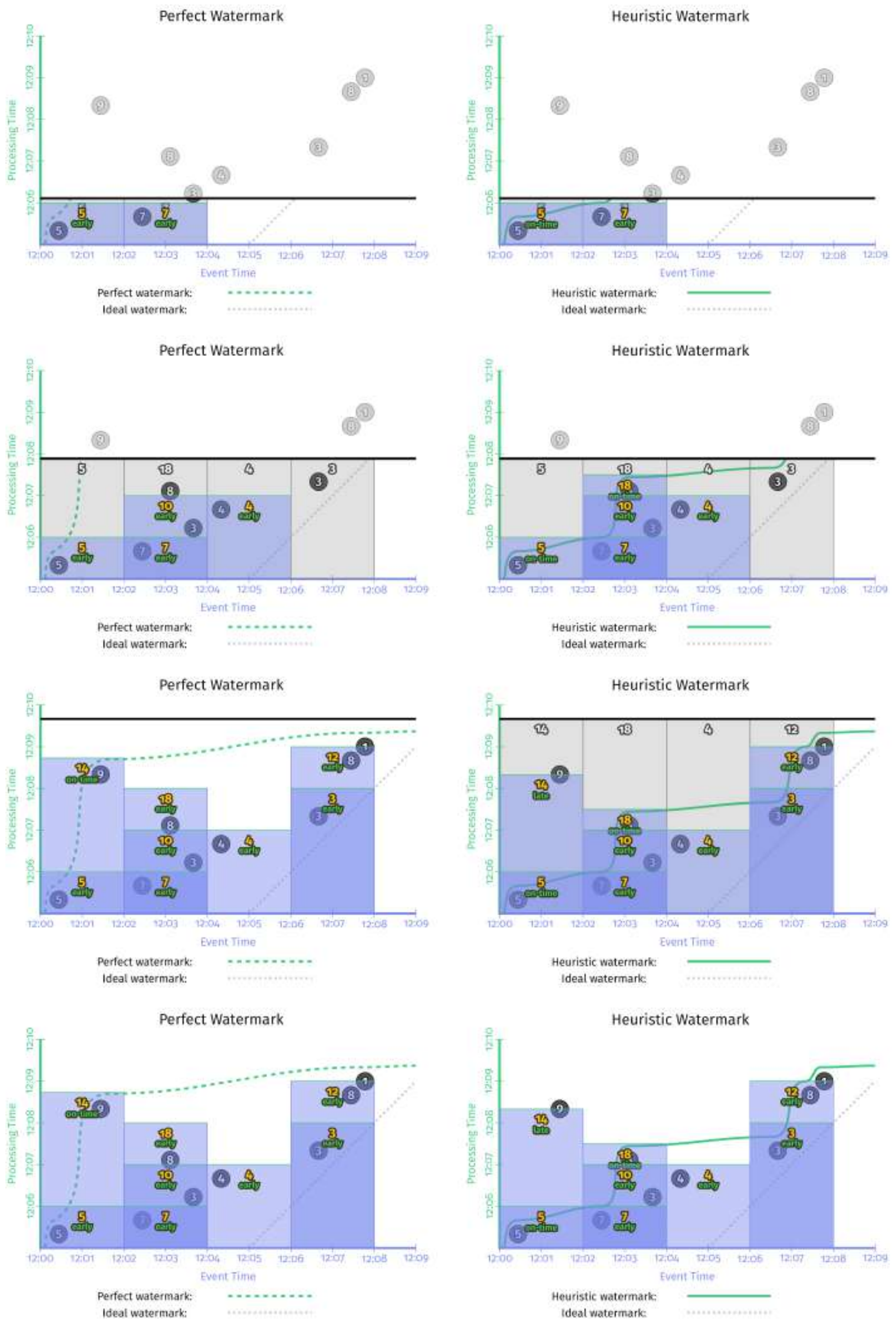


Figure 2-11. Windowed summation on a streaming engine with early, on-time, and late firings

This version has two clear improvements over Figure 2-9:

- For the "watermarks too slow" case in the second window, [12:02, 12:04): we now provide periodic early updates once per minute. The difference is most stark in the perfect watermark case, for which time-to-first-output is reduced from almost seven minutes down to three and a half; but it's also clearly improved in the heuristic case, as well. Both versions now provide steady refinements over time (panes with values 7, 10, then 18), with relatively minimal latency between the input becoming complete and materialization of the final output pane for the window.
- For the "heuristic watermarks too fast" case in the first window, [12:00, 12:02): when the value of 9 shows up late, we immediately incorporate it into a new, corrected pane with value of 14.

One interesting side effect of these new triggers is that they effectively normalize the output pattern between the perfect and heuristic watermark versions. Whereas the two versions in Figure 2-10 were starkly different, the two versions here look quite similar. They also look much more similar to the various repeated update version from Figures 2-6 through 2-8, with one important difference: thanks to the use of the watermark trigger, we can also reason about input completeness in the results we generate with the early/on-time/late trigger. This allows us to better handle use cases that care about *missing data*, like outer joins, anomaly detection, and so on.

The biggest remaining difference between the perfect and heuristic early/on-time/late versions at this point is window lifetime bounds. In the perfect watermark case, we know we'll never see any more data for a window after the watermark has passed the end of it, hence we can drop all of our state for the window at that time. In the heuristic watermark case, we still need to hold on to the state for a window for some amount of time to account for late data. But as of yet, our system doesn't have any good way of knowing just how long state needs to be kept around for each window. That's where *allowed lateness* comes in.

When: Allowed Lateness (i.e., Garbage Collection)

Before moving on to our last question ("*How* do refinements of results relate?"), I'd like to touch on a practical necessity within long-lived, out-of-order stream processing systems: garbage collection. In the heuristic watermarks example in Figure 2-11, the persistent state for each window lingers around for the entire lifetime of the example; this is necessary to allow us to appropriately deal with late data when/if they arrive. But while it would be great to be able to keep around all of our persistent state until the end of time, in reality, when dealing with an unbounded data source, it's often not practical to keep state (including metadata) for a given window indefinitely;

we'll eventually run out of disk space (or at the very least tire of paying for it, as the value for older data diminishes over time).

As a result, any real-world out-of-order processing system needs to provide some way to bound the lifetimes of the windows it's processing. A clean and concise way of doing this is by defining a horizon on the allowed lateness within the system; that is, placing a bound on how late any given *record* may be (relative to the watermark) for the system to bother processing it; any data that arrives after this horizon are simply dropped. After you've bounded how late individual data may be, you've also established precisely how long the state for windows must be kept around: until the watermark exceeds the lateness horizon for the end of the window. But in addition, you've also given the system the liberty to immediately drop any data later than the horizon as soon as they're observed, which means the system doesn't waste resources processing data that no one cares about.

Measuring Lateness

It might seem a little odd to be specifying a horizon for handling late data using the very metric that resulted in the late data in the first place (i.e., the heuristic watermark). And in some sense it is. But of the options available, it's arguably the best. The only other practical option would be to specify the horizon in processing time (e.g., keep windows around for 10 minutes of processing time after the watermark passes the end of the window), but using processing time would leave the garbage collection policy vulnerable to issues within the pipeline itself (e.g., workers crashing, causing the pipeline to stall for a few minutes), which could lead to windows that didn't actually have a chance to handle late data that they otherwise should have. By specifying the horizon in the event-time domain, garbage collection is directly tied to the actual progress of the pipeline, which decreases the likelihood that a window will miss its opportunity to handle late data appropriately.

Note however, that not all watermarks are created equal. When we speak of watermarks in this book, we generally refer to *low* watermarks, which pessimistically attempt to capture the event time of the *oldest* unprocessed record the system is aware of. The nice thing about dealing with lateness via low watermarks is that they are resilient to changes in event-time skew; no matter how large the skew in a pipeline may grow, the low watermark will always track the oldest outstanding event known to the system, providing the best guarantee of correctness possible.

In contrast, some systems may use the term "watermark" to mean other things. For example, watermarks in Spark Structured Streaming are *high* watermarks, which optimistically track the event time of the *newest* record the system is aware of. When dealing with lateness, the system is free to garbage collect any window older than the high watermark adjusted by some user-specified lateness threshold. In other words, the system allows you to specify the maximum amount of event-time skew you expect to see in your pipeline, and then throws away any data outside of that skew window.

This can work well if skew within your pipeline remains within some constant delta, but is more prone to incorrectly discarding data than low watermarking schemes.

Because the interaction between allowed lateness and the watermark is a little subtle, it's worth looking at an example. Let's take the heuristic watermark pipeline from Example 2-7/Figure 2-11 and add in Example 2-8 a lateness horizon of one minute (note that this particular horizon has been chosen strictly because it fits nicely into the diagram; for real-world use cases, a larger horizon would likely be much more practical):

Example 2-8. Early/on-time/late firings with allowed lateness

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(AlignedDelay(ONE_MINUTE))
                   .withLateFirings(AfterCount(1)))
               .withAllowedLateness(ONE_MINUTE))
 .apply(Sum.integersPerKey());
```

The execution of this pipeline would look something like Figure 2-12, in which I've added the following features to highlight the effects of allowed lateness:

- The thick black line denoting the current position in processing time is now annotated with ticks indicating the lateness horizon (in event time) for all active windows.
- When the watermark passes the lateness horizon for a window, that window is closed, which means that all state for the window is discarded. I leave around a dotted rectangle showing the extent of time (in both domains) that the window covered when it was closed, with a little tail extending to the right to denote the lateness horizon for the window (for contrasting against the watermark).
- For this diagram only, I've added an additional late datum for the first window with value 6. The 6 is late, but still within the allowed lateness horizon and thus is incorporated into an updated result with value 11. The 9, however, arrives beyond the lateness horizon, so it is simply dropped.

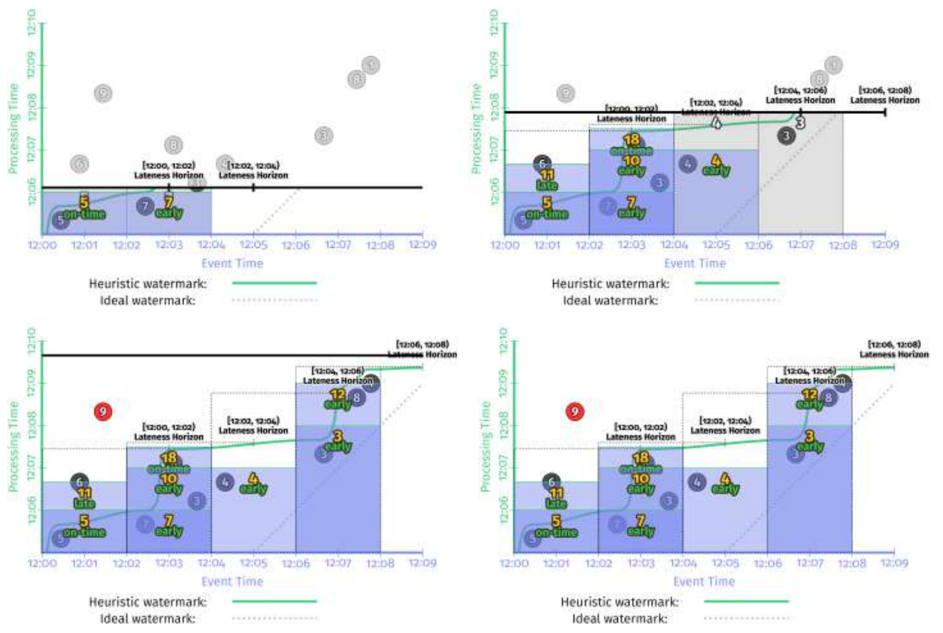


Figure 2-12. Allowed lateness with early/on-time/late firings

Two final side notes about lateness horizons:

- To be absolutely clear, if you happen to be consuming data from sources for which perfect watermarks are available, there's no need to deal with late data, and an allowed lateness horizon of zero seconds will be optimal. This is what we saw in the perfect watermark portion of Figure 2-10.
- One noteworthy exception to the rule of needing to specify lateness horizons, even when heuristic watermarks are in use, would be something like computing global aggregates over all time for a tractably finite number of keys (e.g., computing the total number of visits to your site over all time, grouped by web browser family). In this case, the number of active windows in the system is bounded by the limited keyspace in use. As long as the number of keys remains manageably low, there's no need to worry about limiting the lifetime of windows via allowed lateness.

Practicality sated, let's move on to our fourth and final question.

How: Accumulation

When triggers are used to produce multiple panes for a single window over time, we find ourselves confronted with the last question: "*How* do refinements of results relate?" In the examples we've seen so far, each successive pane is built upon the one immediately preceding it. However, there are actually three[8] different modes of accumulation:[9]

Discarding
: Every time a pane is materialized, any stored state is discarded. This means that each successive pane is independent from any that came before. Discarding mode is useful when the downstream consumer is performing some sort of accumulation itself; for example, when sending integers into a system that expects to receive deltas that it will sum together to produce a final count.

Accumulating
: As in Figures 2-6 through 2-11, every time a pane is materialized, any stored state is retained, and future inputs are accumulated into the existing state. This means that each successive pane builds upon the previous panes. Accumulating mode is useful when later results can simply overwrite previous results, such as when storing output in a key/value store like HBase or Bigtable.

Accumulating and retracting
: This is like accumulating mode, but when producing a new pane, it also produces independent retractions for the previous pane(s). Retractions (combined with the new accumulated result) are essentially an explicit way of saying "I previously told you the result was *X*, but I was wrong. Get rid of the *X* I told you last time, and replace it with *Y*." There are two cases for which retractions are particularly helpful:

    - When consumers downstream are *regrouping data by a different dimension*, it's entirely possible the new value may end up keyed differently from the previous value and thus end up in a different group. In that case, the new value can't just overwrite the old value; you instead need the retraction to remove the old value
    - When *dynamic windows* (e.g., sessions, which we look at more closely in a few moments) are in use, the new value might be replacing more than one

8 You might note that there should logically be a fourth mode: discarding and retracting. That mode isn't terribly useful in most cases, so I don't discuss it further here.

9 In retrospect, it probably would have been clearer to choose a different set of names that are more oriented toward the observed nature of data in the materialized stream (e.g., "output modes") rather than names describing the state management semantics that yield those data. Perhaps: discarding mode → delta mode, accumulating mode → value mode, accumulating and retracting mode → value and retraction mode? However, the discarding/accumulating/accumulating and retracting names are enshrined in the 1.x and 2.x lineages of the Beam Model, so I don't want to introduce potential confusion in the book by deviating. Also, it's very likely accumulating modes will blend into the background more with Beam 3.0 and the introduction of sink triggers; more on this when we discuss SQL later in this book.

previous window, due to window merging. In this case, it can be difficult to determine from the new window alone which old windows are being replaced. Having explicit retractions for the old windows makes the task straightforward.

The different semantics for each group are somewhat clearer when seen side-by-side. Consider the two panes for the second window (the one with event-time range [12:06, 12:08)) in Figure 2-11 (the one with early/on-time/late triggers). Table 2-1 shows what the values for each pane would look like across the three accumulation modes (with *accumulating* mode being the specific mode used in Figure 2-11 itself).

Table 2-1. Comparing accumulation modes using the second window from Figure 2-11

	Discarding	Accumulating	Accumulating & Retracting
Pane 1: inputs=[3]	3	3	3
Pane 2: inputs=[8, 1]	9	12	12, –3
Value of final normal pane	9	12	12
Sum of all panes	12	15	12

Let's take a closer look at what's happening:

Discarding
Each pane incorporates only the values that arrived during that specific pane. As such, the final value observed does not fully capture the total sum. However, if you were to sum all of the independent panes themselves, you would arrive at a correct answer of 12. This is why discarding mode is useful when the downstream consumer itself is performing some sort of aggregation on the materialized panes.

Accumulating
As in Figure 2-11, each pane incorporates the values that arrived during that specific pane, plus all of the values from previous panes. As such, the final value observed correctly captures the total sum of 12. If you were to sum up the individual panes themselves, however, you'd effectively be double-counting the inputs from pane 1, giving you an incorrect total sum of 15. This is why accumulating mode is most useful when you can simply overwrite previous values with new values: the new value already incorporates all of the data seen thus far.

Accumulating and retracting
Each pane includes both a new accumulating mode value as well as a retraction of the previous pane's value. As such, both the last value observed (excluding retractions) as well as the total sum of all materialized panes (including retractions) provide you with the correct answer of 12. This is why retractions are so powerful.

Example 2-9 demonstrates discarding mode in action, illustrating the changes we would make to Example 2-7:

Example 2-9. Discarding mode version of early/on-time/late firings

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(AlignedDelay(ONE_MINUTE))
                   .withLateFirings(AtCount(1)))
               .discardingFiredPanes())
  .apply(Sum.integersPerKey());
```

Running again on a streaming engine with a heuristic watermark would produce output like that shown in Figure 2-13.

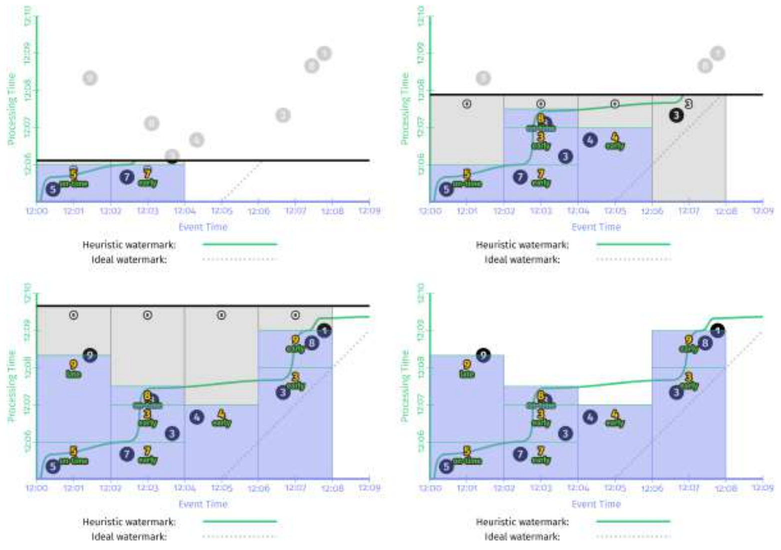


Figure 2-13. Discarding mode version of early/on-time/late firings on a streaming engine

Even though the overall shape of the output is similar to the accumulating mode version from Figure 2-11, note how none of the panes in this discarding version overlap. As a result, each output is independent from the others.

If we want to look at retractions in action, the change would be similar, as shown in Example 2-10. Figure 2-14 depicts the results.

Example 2-10. Accumulating and retracting mode version of early/on-time/late firings

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(AlignedDelay(ONE_MINUTE))
                   .withLateFirings(AtCount(1)))
               .accumulatingAndRetractingFiredPanes())
  .apply(Sum.integersPerKey());
```

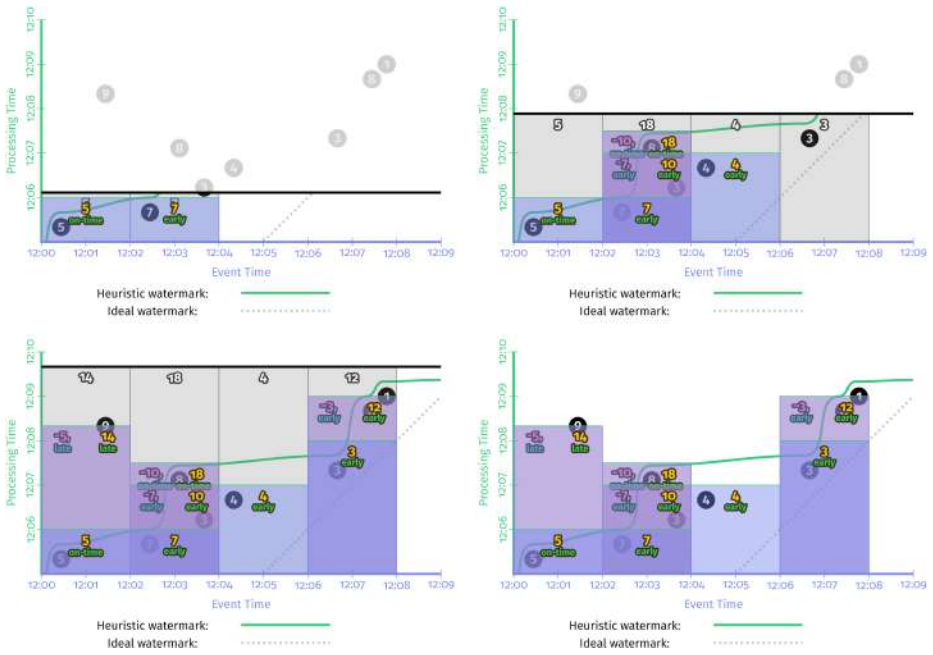


Figure 2-14. Accumulating and retracting mode version of early/late firings on a streaming engine

Because the panes for each window all overlap, it's a little tricky to see the retractions clearly. The retractions are indicated in red, which combines with the overlapping blue panes to yield a slightly purplish color. I've also horizontally shifted the values of the two outputs within a given pane slightly (and separated them with a comma) to make them easier to differentiate.

Figure 2-15 combines the final frames of Figures 2-9, 2-11 (heuristic only), and 2-14 side-by-side, providing a nice visual contrast of the three modes.

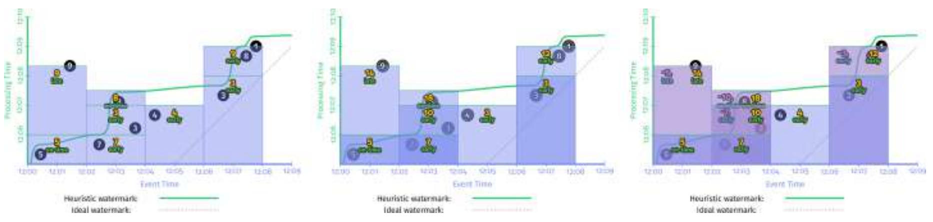


Figure 2-15. Side-by-side comparison of accumulation modes

As you can imagine, the modes in the order presented (discarding, accumulating, accumulating and retracting) are each successively more expensive in terms of storage and computation costs. To that end, choice of accumulation mode provides yet another dimension for making trade-offs along the axes of correctness, latency, and cost.

Summary

With this chapter complete, you now understand the basics of robust stream processing and are ready to go forth into the world and do amazing things. Of course, there are eight more chapters anxiously waiting for your attention, so hopefully you won't go forth like right now, this very minute. But regardless, let's recap what we've just covered, lest you forget any of it in your haste to amble forward. First, the major concepts we touched upon:

Event time versus processing time
: The all-important distinction between when events occurred and when they are observed by your data processing system.

Windowing
: The commonly utilized approach to managing unbounded data by slicing it along temporal boundaries (in either processing time or event time, though we narrow the definition of windowing in the Beam Model to mean only within event time).

Triggers
: The declarative mechanism for specifying precisely when materialization of output makes sense for your particular use case.

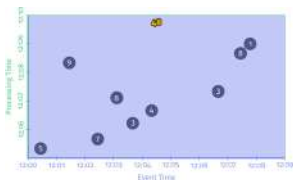
Watermarks
: The powerful notion of progress in event time that provides a means of reasoning about completeness (and thus missing data) in an out-of-order processing system operating on unbounded data.

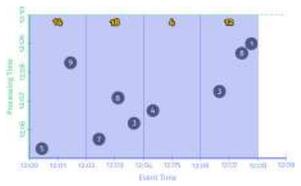
Accumulation
: The relationship between refinements of results for a single window for cases in which it's materialized multiple times as it evolves.

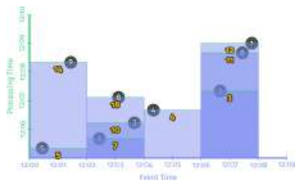
Second, the four questions we used to frame our exploration:

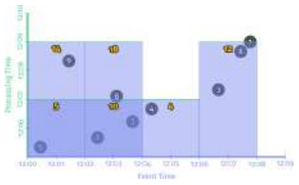
- *What* results are calculated? = transformations.
- *Where* in event time are results calculated? = windowing.
- *When* in processing time are results materialized? = triggers plus watermarks.
- *How* do refinements of results relate? = accumulation.

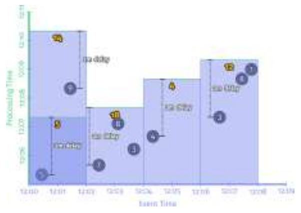
Third, to drive home the flexibility afforded by this model of stream processing (because in the end, that's really what this is all about: balancing competing tensions like correctness, latency, and cost), a recap of the major variations in output we were able to achieve over the same dataset with only a minimal amount of code change:

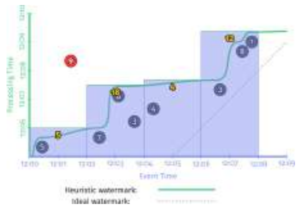

Integer summation
Example 2-1 / Figure 2-3


Integer summation
Fixed windows batch
Example 2-2 / Figure 2-5

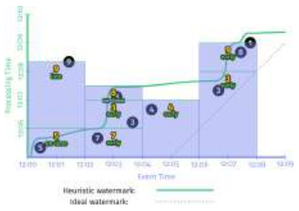

Integer summation
Fixed windows streaming
Repeated per-record trigger
Example 2-3 / Figure 2-6


Integer summation
Fixed windows streaming
Repeated aligned-delay trigger
Example 2-4 / Figure 2-7

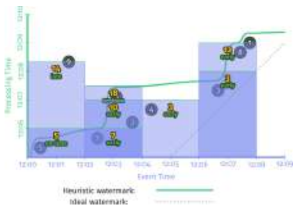

Integer summation
Fixed windows streaming
Repeated unaligned-delay trigger
Example 2-5 / Figure 2-8



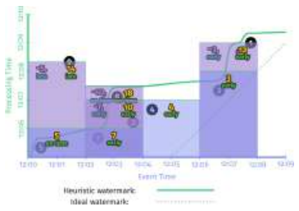
Integer summation
Fixed windows streaming
Heuristic watermark trigger
Example 2-6 / Figure 2-10



Integer summation
Fixed windows streaming
Early/on-time/late trigger
Discarding
Example 2-9 / Figure 2-13



Integer summation
Fixed windows streaming
Early/on-time/late trigger
Accumulating
Example 2-7 / Figure 2-11



Integer summation
Fixed windows streaming
Early/on-time/late trigger
Accumulating and Retracting
Example 2-10 / Figure 2-14

All that said, at this point, we've really looked at only one type of windowing: fixed windowing in event time. As we know, there are a number of dimensions to windowing, and I'd like to touch upon at least two more of those before we call it day with the Beam Model. First, however, we're going to take a slight detour to dive deeper into the world of watermarks, as this knowledge will help frame future discussions (and be fascinating in and of itself). Enter Slava, stage right...

About the Authors

Tyler Akidau is a senior staff software engineer at Google, where he is the technical lead for the Data Processing Languages & Systems group, responsible for Google's Apache Beam efforts, Google Cloud Dataflow, and internal data processing tools like Google Flume, MapReduce, and MillWheel. Tyler is also a founding member of the Apache Beam PMC. Though deeply passionate and vocal about the capabilities and importance of stream processing, he is also a firm believer in batch and streaming as two sides of the same coin, with the real endgame for data processing systems being the seamless merging between the two. He is the author of the "Dataflow Model" paper and the "Streaming 101" and "Streaming 102" articles on the O'Reilly website. His preferred mode of transportation is by cargo bike, with his two young daughters in tow.

Slava Chernyak is a senior software engineer at Google Seattle. Slava spent more than six years working on Google's internal massive-scale streaming data processing systems and has since become involved with designing and building Windmill, Google Cloud Dataflow's next-generation streaming backend, from the ground up. Slava is passionate about making massive-scale stream processing available and useful to a broader audience. When he is not working on streaming systems, Slava is out enjoying the natural beauty of the Pacific Northwest.

Reuven Lax is a senior staff software engineer at Google, Seattle, and has spent the past ten years helping to shape Google's data processing and analysis strategy. For much of that time he has focused on Google's low-latency, streaming data processing efforts, first as a long-time member and lead of the MillWheel team, and more recently founding and leading the team responsible for Windmill, the next-generation stream processing engine powering Google Cloud Dataflow. He is also a Beam PMC member. He's very excited to bring Google's data processing experience to the world at large and proud to have been a part of publishing both the "MillWheel" paper in 2013 and the "Dataflow Model" paper in 2015. When not at work, Reuven enjoys swing dancing, rock climbing, and exploring new parts of the world.

Colophon

The animal on the cover of *Streaming Systems* is a brown trout (*Salmo trutta*) a species of medium-sized fish native to northern Europe but now distributed across the globe. Brown trout generally weigh about 2 pounds and grow to a length of 16–31 inches. They have an overall shiny brown color with many dark spots over their upper body.

Brown trout feed mostly on aquatic invertebrates although larger members of the species have been known to prey on other fish. During spawning, the female brown

trout produces thousands of eggs. It takes 3–4 years for a brown trout to reach maturity.

Popular with anglers, brown trout were introduced into lakes and rivers throughout the world during the 19th and early 20th centuries. To this day, brown trout are farmed commercially and stocked for recreational fishing. Brown trout are edible and can be prepared in several ways, including grilling, frying, and smoking.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Karen Montgomery. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.