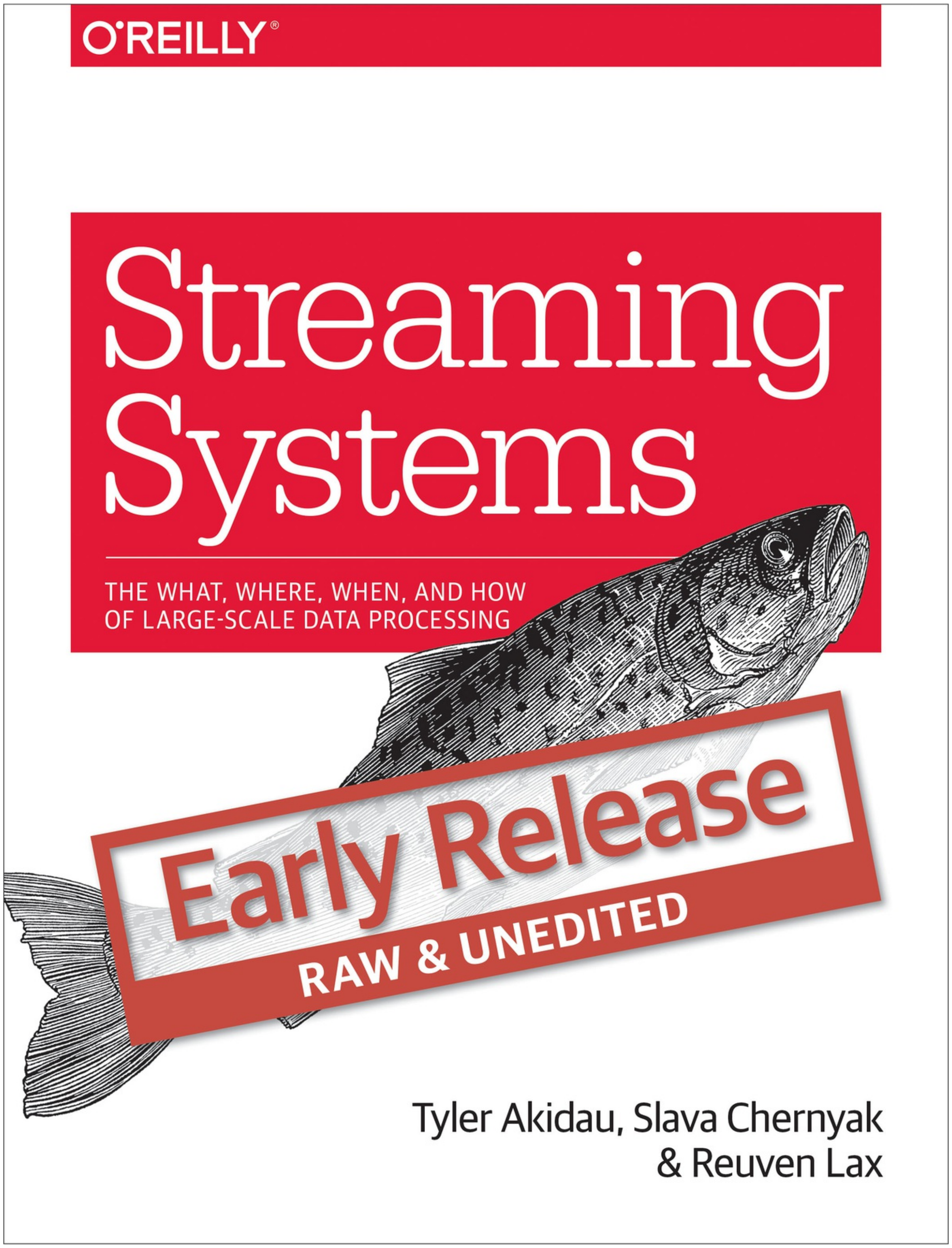
O'REILLY
Streaming Systems
THE WHAT, WHERE, WHEN, AND HOW OF LARGE-SCALE DATA PROCESSING
Early Release
RAW & UNEDITED
Tyler Akidau, Slava Chernyak & Reuven Lax

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.

- Editor: Marie Beaugureau
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

- January -4712: First Edition

Revision History for the First Edition

- 2017-05-23: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=0636920073994 for release details.

063-6-920-07399-4

[FILL IN]

Streaming Systems

First Edition

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

Tyler Akidau, Slava Chernyak, and Reuven Lax

Chapter 1. Why Stream Processing?

Streaming data processing is a big deal in Big Data these days, and for good reasons. Amongst them:

- Businesses crave ever more timely 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 allow myself to think that's maybe in some small part due to the solid dose of goading I originally served up in my Streaming 101 and 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 that still haven't gotten the memo; this book is an extended attempt at getting these points across.

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

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

- **Capabilities** - I'll remark on the oft-perceived shortcomings of streaming systems. I'll 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'll 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 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 (e.g. 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 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: a type of data processing engine that is designed with infinite datasets in mind. Nothing more. (For completeness, it's perhaps worth calling out that this definition includes both true streaming as well as micro-batch[1]implementations)

As to other common uses of the term "streaming", here are a few that I hear regularly, each presented with the more precise, descriptive terms that I suggest we as a community should try to adopt:

1. **Unbounded Data**: A type of ever-growing, essentially infinite dataset. These are often referred to as "streaming" data or "streams". However, the terms streaming or batch are problematic when applied to datasets, because as noted above, they imply the use of a certain type of execution engine for processing those datasets. The key distinction between the two types of datasets in question is, in reality, their finiteness, and it's

thus preferable to characterize them by terms that capture this distinction. As such, I will refer to infinite "streaming" datasets as unbounded data, and finite "batch" datasets as bounded data.

2. **Unbounded or Continuous Data Processing**: An ongoing mode of data processing, applied to the aforementioned type of unbounded data. As much as I personally like the use of the term "streaming" to describe this type of data processing, its use in this context again implies the employment of a streaming execution engine, which is at best misleading; repeated runs of batch engines have been used to process unbounded data since batch systems were first conceived (and conversely, well-designed streaming systems are more than capable of handling "batch" workloads over bounded data). As such, for the sake of clarity, I will prefer to simply refer to this as continuous data processing.

3. Low-Latency, Approximate, and/or Speculative Results: These types of results are most often associated with streaming engines. The fact that batch systems have traditionally not been designed with low-latency or speculative results in mind is a historical artifact, and nothing more. And of course, batch engines are perfectly capable of producing approximate results if instructed to. Thus, as with the terms above, it's far better describing these results as what they are (low-latency, approximate, and/or speculative) than by how they have historically been manifested (via streaming engines).

From here on out, any time I use the term "streaming", you can safely assume I mean an execution engine designed for unbounded datasets, and nothing more. When I mean any of the other terms above, I will explicitly say unbounded data, continuous data processing, or low-latency / approximate / speculative results. I encourage others to take a similar stance.

On The Greatly Exaggerated Limitations of Streaming

Next up, let’s 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/speculative results, often in conjunction with a more capable batch system to provide eventually correct results, i.e. 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 it's 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[2], 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.

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, i.e. to beat batch at its own game, you really only need two things:

1. **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; I seriously cannot believe that at-most-once processing is still a thing, but it is.

   To reiterate, because this point is important: strong consistency is required for exactly-once processing, 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. Given the amount of quality information on this topic in the literature and elsewhere, I won’t be covering it any further in this book.

2. **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 next few chapters, and really the bulk of this book, explaining and focusing on this point.

   To begin with, we’ll get a basic understanding of the important concept of time domains, after which we’ll take a deeper look at what I mean by unbounded, unordered data of varying event-time skew. We’ll 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 vs. 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 we care about:

- **Event time**, which is the time at which events actually occurred.
- **Processing time**, which 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 non-zero, 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:

- Shared resource limitations, like network congestion, network partitions, or shared CPU in a non-dedicated environment.
- Software causes such as distributed system logic, contention, etc.
- 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.

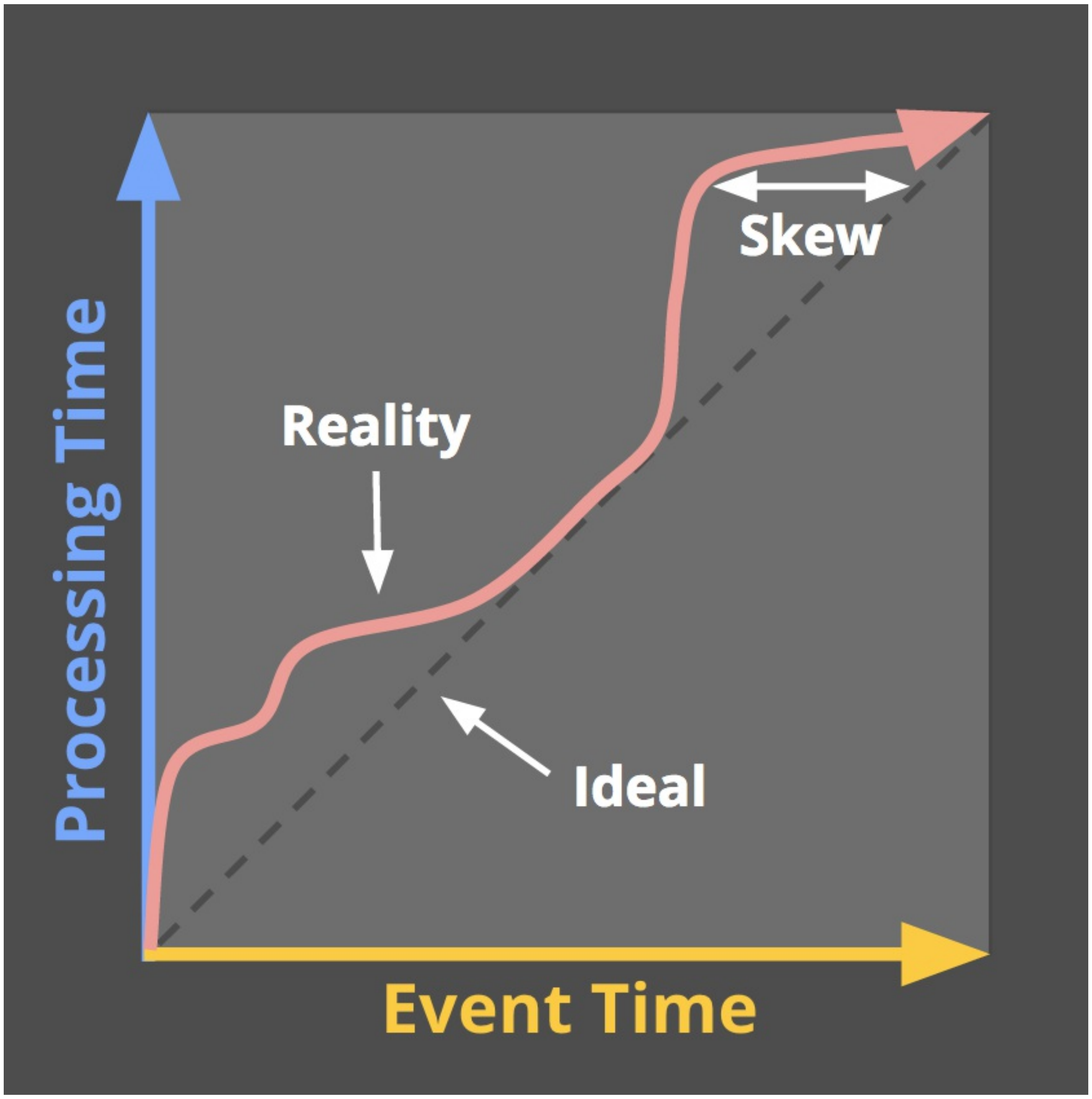

Figure 1-1. Example Time Domain Mapping. The X axis represents event time completeness in the system, i.e. the time X in event time up to which all data with event times less than X have been observed. The Y axis[3] represents the progress of processing time, i.e. normal clock time as observed by the data processing system as it executes.

The black dashed line with slope of one 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 towards the ideal in the middle, then lags again a bit towards the end. At first glance, there are two types of skew visible in this diagram, each in different time domains:

- **Processing time lag** - 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 they were processed. This is the perhaps the more natural and intuitive of the two skews.

- **Event time skew** - 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[4]. The important takeaway regarding lag/skew is this: since the overall mapping between event time and processing time is not static (i.e. the lag/skew can vary arbitrarily over time), this means 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 most 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'll discuss windowing in great depth below, 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'll look at this problem in more detail in a number of examples below, as well as in the following post.

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 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 may 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 a convenient optimization rather than a semantic necessity.

Before getting into specifics about what such a system might look like, we’ll spend the next chapter covering one more useful piece of background: common data processing patterns.

[1] For those of you who aren’t familiar with micro-batch systems, they are streaming systems which use repeated executions of a batch processing engine to process unbounded data. Spark Streaming is the canonical example in the industry.

[2] 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 which 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.

[3] 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).

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

Chapter 2. Data Processing Patterns

Thanks to Chapter 1, we now have enough background established to start looking at the core types of usage patterns common across bounded and unbounded data processing today. We’ll 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 micro-batch in with streaming, since 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 the diagram below, 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:

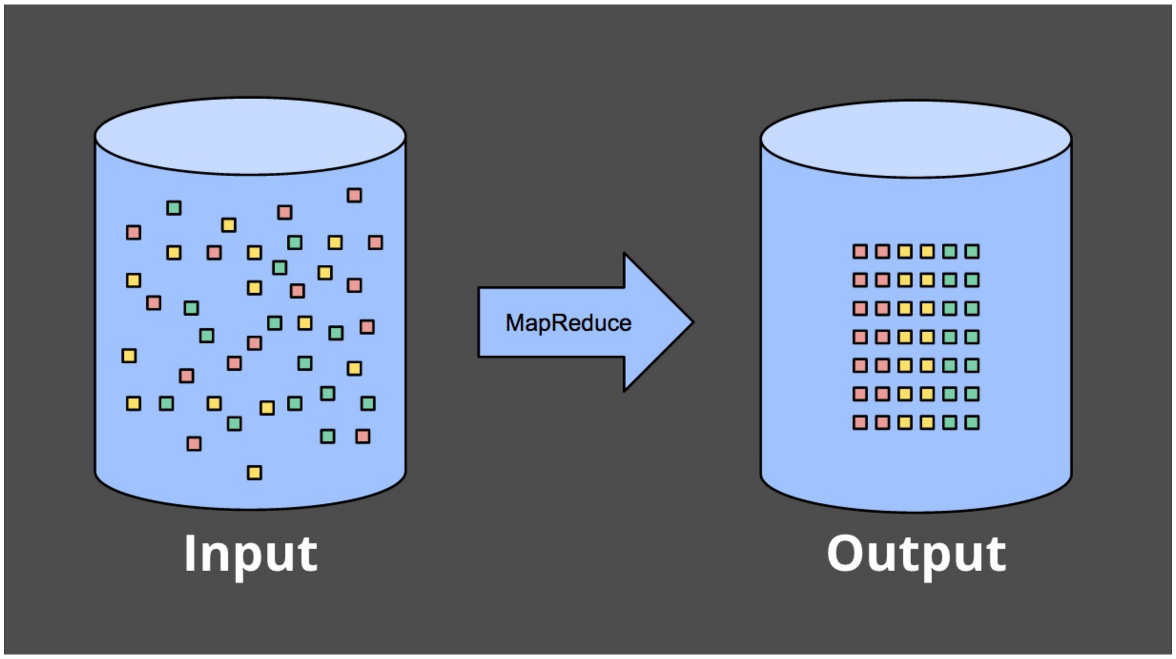

Figure 2-1. Bounded Data Processing with a Classic Batch Engine. A finite pool of unstructured data on the left is run through a data processing engine, resulting in corresponding structured data on the right.

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, starting with the approaches used with traditional batch engines, and then ending up with the approaches one can take with a system designed for unbounded data, such as most streaming or micro-batch engines.

Unbounded Data: Batch

Batch engines, though not explicitly designed with unbounded data in mind, have been used to process unbounded datasets since batch systems were first conceived. As one 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 sized windows, then processing each of those windows as a separate, bounded data source (sometimes also called tumbling windows). Particularly for input sources like logs, where 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, since 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 may be necessary (e.g., delaying processing until you’re sure all events have been collected, or re-processing the entire batch for a given window whenever data arrive late).

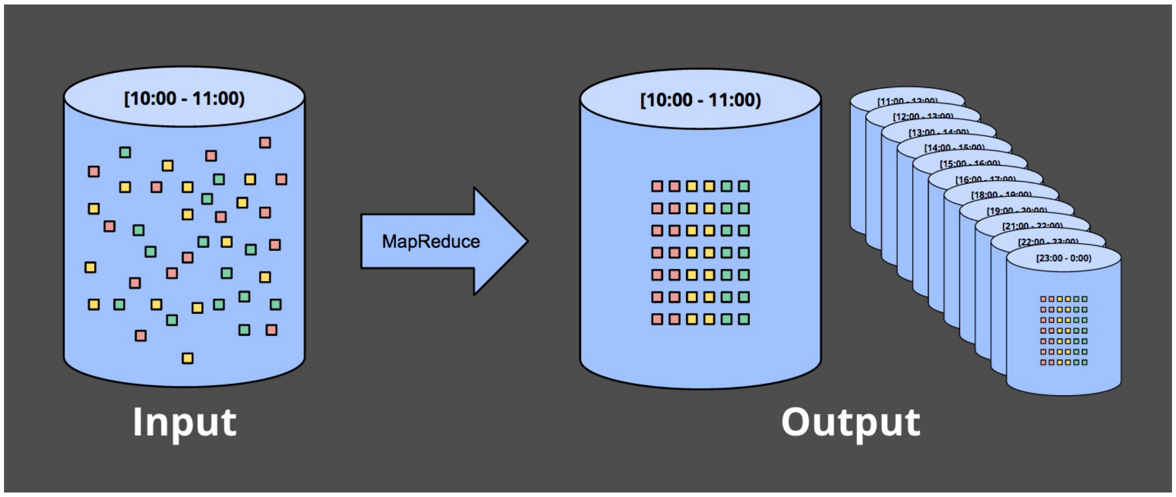

Figure 2-2. Unbounded Data Processing via Ad Hoc Fixed Windows with a Classic Batch Engine. An unbounded dataset is collected up front into finite, fixed-size windows of bounded data that are then processed via successive runs a of classic batch engine.

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 the diagram below. The number of splits can be reduced 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.

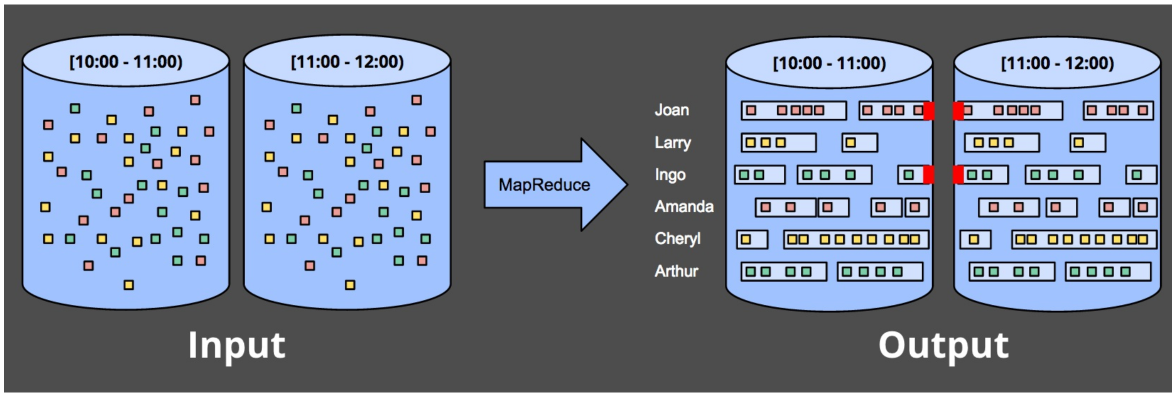

Figure 2-3. Unbounded Data Processing into Sessions via Ad Hoc Fixed Windows with a Classic Batch Engine. An unbounded dataset is collected up front into finite, fixed-size windows of bounded data that are then subdivided into dynamic session windows via successive runs a of classic batch engine.

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’ll 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 above, for many real-world, distributed input sources, you not only find yourself dealing with unbounded data, but also data that are:

- Highly unordered with respect to event times, meaning 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 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 one 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
- Windowing by Event Time

We'll now spend a little bit of time looking at each of these approaches.

Time-Agnostic

Time-agnostic processing is used in cases where time is essentially irrelevant, i.e., all relevant logic is data driven. Since 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, for those of you that 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'll 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), won't spend much more time on it beyond that.

Filtering

A very basic form of time-agnostic processing is filtering. Imagine 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. Since 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.

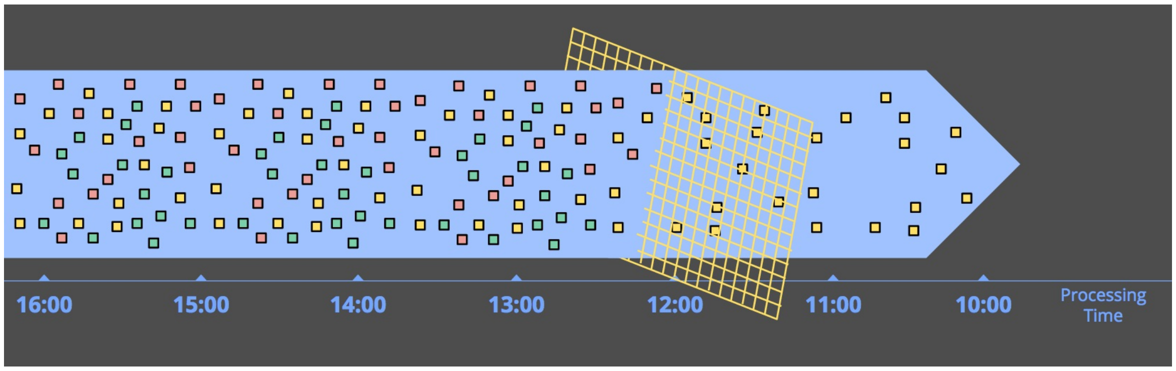

Figure 2-4. Filtering Unbounded Data. A collection of data (flowing left to right) of varying types

is filtered into a homogeneous collection containing a single type.

Inner Joins

Another time-agnostic example is an inner-join. When joining two unbounded data sources, if you only care 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 once 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.)

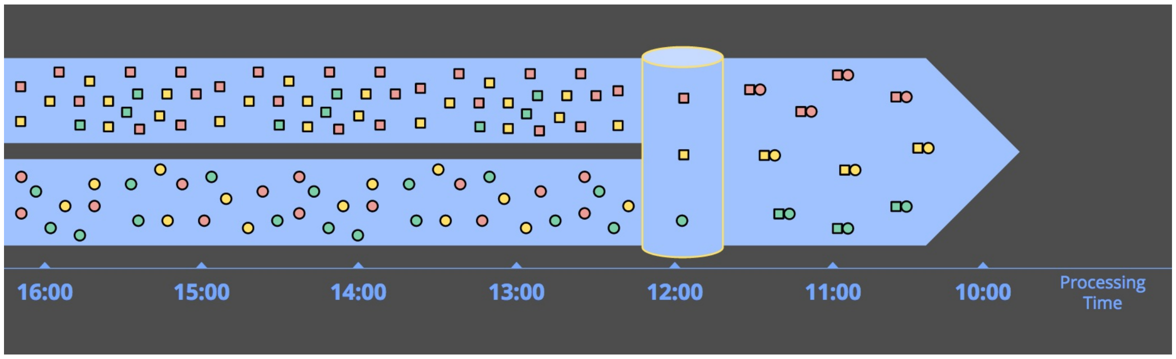

Figure 2-5. Performing an Inner Join on Unbounded Data. Joins are produced when matching elements from both sources are observed.

Switching semantics to some sort of outer join introduces the data completeness problem we've talked about: once 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 have 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 below.

Approximation Algorithms

The second major category of approaches is approximation algorithms, such as approximate Top-N, streaming K-means, etc. 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. 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.

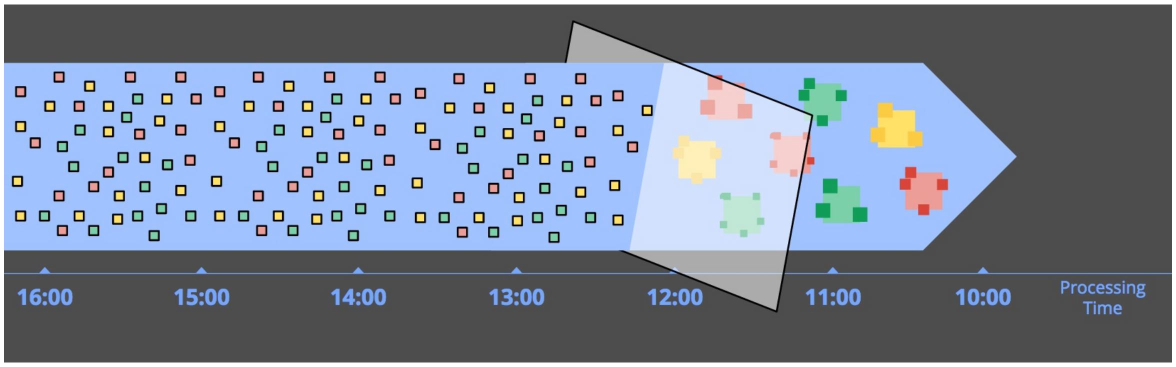

Figure 2-6. Computing Approximations on Unbounded Data. Data are run through a complex algorithm, yielding output data that look more or less like the desired result on the other side.

It's worth noting, these algorithms typically do have some element of time in their design (e.g., some sort of built-in decay). And since 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, since we only touched on it briefly above. 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. The diagram below shows three different windowing patterns:

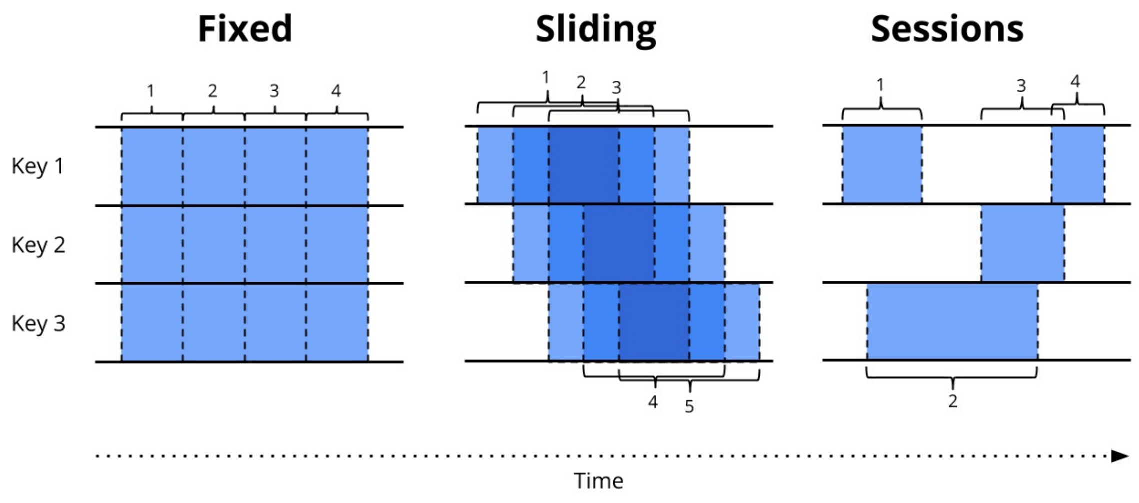

Figure 2-7. Example Windowing Strategies. Each example is shown for three different keys, highlighting the difference between aligned windows (which apply across all the data) and unaligned windows (which apply across a subset of the data).

- **Fixed windows (aka tumbling windows)**: We discussed fixed windows earlier. Fixed windows slice up time into segments with a fixed-size temporal length. Typically (as in the diagram below), the segments for fixed windows are applied uniformly across the entire dataset, which is an example 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, since they vary across the data. Fixed windows are also sometimes referred to as tumbling windows.

- **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, then 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 only looks at subsets of the data over time. As with fixed windows, sliding windows are typically aligned, though may be unaligned as a performance optimization in certain use cases. Note that the sliding windows in the diagram above are drawn as they are to give a sense of sliding motion; in reality, all five windows would apply across the entire dataset. Sliding windows are sometimes also referred to as hopping windows.

- **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, since sessions are practically never identical across different subsets of data (e.g., different users).

The two domains of time we discussed above (processing time and event time) are essentially the two we care about[1]. Windowing makes sense in both domains, so we'll look at each in detail and see how they differ. Since processing time windowing is vastly more common in existing systems, we'll start there.

Windowing by Processing Time

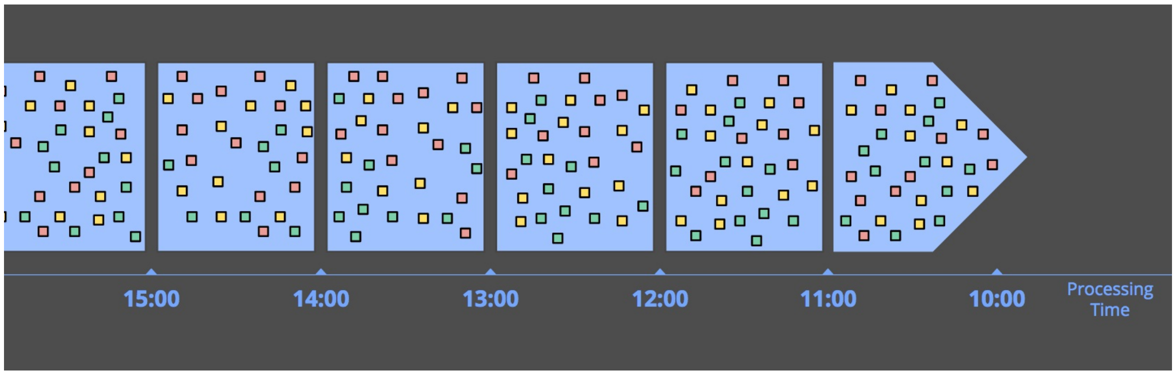

Figure 2-8. Windowing into Fixed Windows by Processing Time. Data are collected into windows based on the order they arrive in the pipeline.

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 up data for five minutes of processing time, after which it would treat all the data it had observed in those five minutes as a window, and send them downstream for processing.

There are a few nice properties of processing time windowing:

- It's **simple**. The implementation is extremely straightforward, since you never worry about shuffling data within time. You just buffer things up as they arrive, and send them downstream when the window closes.

- Judging window **completeness is straightforward**. Since the system has perfect knowledge of whether all inputs for a window have been seen or not, it can make perfect decisions about whether a given window is complete or not. 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. In cases where 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. That means 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 may *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 may start 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. Most data processing systems in use when Streaming 101 was originally published (mid 2015) lacked native support for it (though any system with a decent consistency model, like Hadoop or Spark Streaming 1.x, can act as a reasonable substrate for building such a windowing system). Nearly two years later, I'm happy to say that the world looks very different, with multiple systems, from Flink to Spark to Storm to Apex, natively supporting event time windowing of some sort.

This diagram shows an example of windowing an unbounded source into one-hour fixed windows:

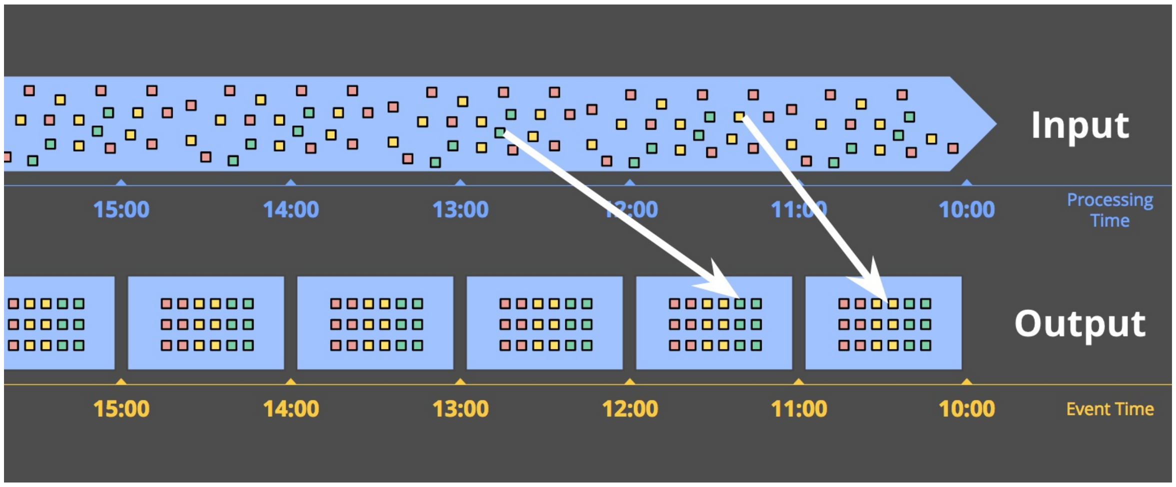

Figure 2-9. Windowing into Fixed Windows by Event Time. Data are collected into windows based on the times they occurred. The white arrows call out example data which arrived in processing time windows that differed from the event time windows to which they belonged.

The solid white lines in the diagram call out two particular data of interest. Those two data both arrived in processing time windows that did not match the event time windows to which they 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 one 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 the Unbounded Data: Batch section):

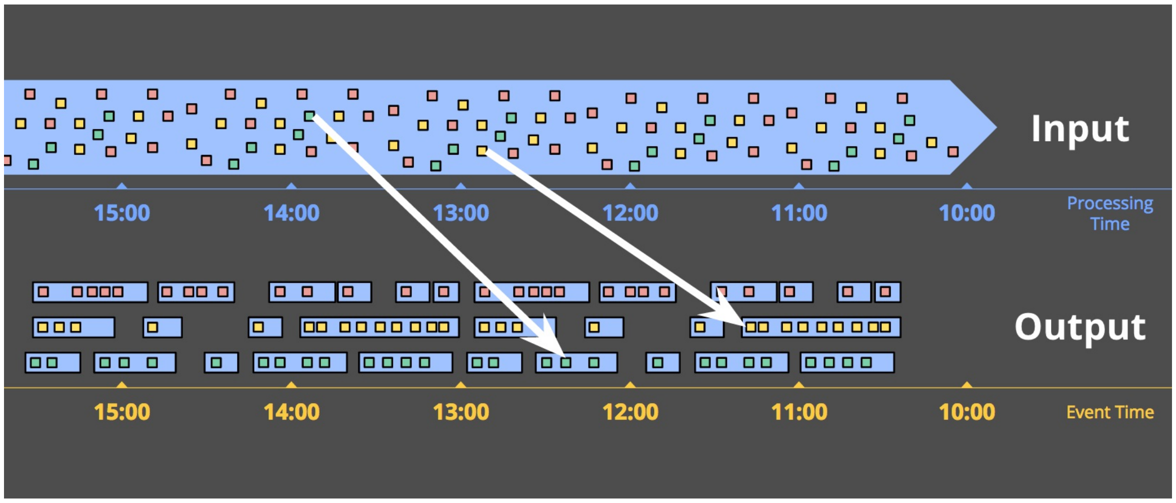

Figure 2-10. Windowing into Session Windows by Event Time. Data are collected into session windows capturing bursts of activity based on the times that the corresponding events occurred. The white arrows again call out the temporal shuffle necessary to put the data into their correct event-time locations.

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 one 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 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 MillWheel’s watermarks (which we’ll talk about more in chapters 3 and 4). But in cases where 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’ll look at next.

Recap

Whew! That was a lot of information. To those of you that have made it this far: you are to be commended! To those who gave up: you are weak! (Just kidding, but they probably won't read this anyway, so it's cool.) By raw word count, we are now 9.6% of the way through the book. Not bad! This seems as reasonable a place as any to step back and recap what we've learned so far.

In the last two chapters, we've:

- Clarified terminology, specifically **narrowing the definition of "streaming"** to apply to execution engines only, while using more descriptive terms like **unbounded data** and **approximate/speculative results** for distinct concepts often categorized under the "streaming" umbrella.

- 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 **towards 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’ll be diving 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’ll 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 post, while additionally exploring new concepts, such as watermarks.

[1] 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 where elements are assigned monotonically increasing timestamps as they arrive at the system. As such, we won’t discuss tuple-based windowing in detail here.

Chapter 3. The *What*, *Where*, *When*, and *How* of Data Processing

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

The last two chapters focused on three main areas: **terminology**, defining precisely what I mean when I use overloaded terms like “streaming”; **batch vs 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 2, 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’ll use snippets of Apache Beam code, coupled with static diagrams[1] to provide a visual representation of the concepts. Apache Beam is a unified programming model for batch and stream processing, with a set of concrete SDKs in various languages (e.g. Java, Python, etc.). Pipelines written with Apache Beam can then be portably run on any of the supported execution engines (e.g. 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 half later, I'm happy to say much has changed, and most of the major systems out there have moved or are moving towards 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 I. 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.

In Chapter 2, 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 may never end. Some simpler examples of windowing strategies are fixed and sliding windows, but more sophisticated types of windowing, such as sessions (where the windows are defined by features of the data themselves, e.g., 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:

- **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'll touch upon the basics of watermarks in this chapter, and then Slava will go super deep on the subject in Chapter 4.

- **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 which arrive late relative to the watermark (e.g., mobile scenarios, where someone's phone records various actions and their event times while the person is offline, then proceeds to upload those events for processing upon regaining connectivity).

- **Accumulation** - An accumulation mode specifies the relationship between multiple results that are observed for the same window. Those results might be completely disjoint, i.e., representing independent deltas over time, or there may 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'll 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, etc. 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 2 (fixed, sliding, and sessions), use cases which 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 one assigns 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 watermarks and triggers. There are infinite variations on this theme, but the most common pattern uses the watermark to delineate when input for a given window is complete, with triggers allowing the specification of early results (for speculative, partial results emitted before the window is complete) and late results (for cases where the watermark is only an estimate of completeness, and more input data may arrive after the watermark claims the input for a given window is complete).

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

Well look at each of these questions in much more detail over the rest of the post. 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* & *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 many of you are likely already familiar with classic batch processing, we're going to start there anyway since it's the foundation on top of which we'll add all of the other concepts.

In the rest of this chapter, we'll look at a single example: computing keyed integer sums over a simple dataset consisting of 10 values. If you want a slightly more pragmatic take on it, you can think of it as calculating an overall score for a team of individuals playing some sort of mobile game by combining together their independent scores. You can imagine it applying similarly well to billing and usage monitoring use cases.

For each example, I'll include a short snippet of Apache Beam Java SDK pseudo-code to make the definition of the pipeline more concrete. It will be pseudo-code in the sense that I'll 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 Java are painfully verbose; I will use simpler names for clarity). Beyond minor things like those (most of which I enumerate explicitly in the epilogue), it's basically real-world Beam code. I'll also provide a link to an actual code walkthrough later on for those that are interested in similar examples they can compile and run themselves.

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

- **PCollections**, which represent datasets (possibly massive ones), across which parallel transformations may be performed (hence the "P" at the beginning of the name).

- **PTransforms**, which are applied to PCollections to create new PCollections. PTransforms may perform element-wise transformations, they may aggregate multiple elements together, or they may be a composite combination of other PTransforms.

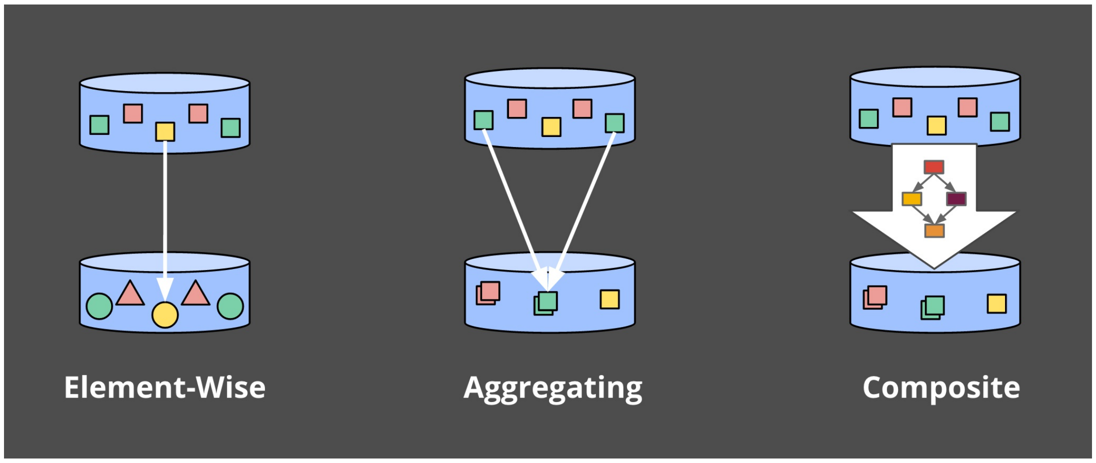

Figure 3-1. Types of transformations.

If you find yourself getting confused, or just want a reference to consult, you might give the [Apache Beam Programming Model docs] a look.

For the purposes of our examples, we'll assume we start out with a 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 of raw data (e.g., log records) from an I/O source, and then transforming it into the PCollection<KV<Team, Integer>> by parsing the log records into appropriate key/value pairs. For the sake of clarity in this first example, I'll include pseudo-code 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 out team/score pairs, and calculates per-team sums of scores, we'd have something like this:

Table 3-1. **Summation Pipeline.** Key/value data are read from an I/O source Team (e.g., String of the team name) as the key and an Integer (e.g., individu member scores) as the value. The values for each key are then summed toge generate per-key sums (e.g., total team score) in the output collection

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

For all the examples to come, after seeing a code snippet describing the pipeline we'll be analyzing, we'll then look at a time-lapse diagram showing specific points in time of the execution of that pipeline over a concrete dataset (we'll also provide a URL for an animated video of the same diagram). More specifically, we'll see what it would look like to execute the pipeline over ten input data 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'll be clearer to keep things simple.

Each animation 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 ascending white line. Inputs are circles, with the number inside the circle representing the value of that specific record. They start out grey, and change color 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, 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 Table 3-1, it would look something like this when executed on a classic batch engine:

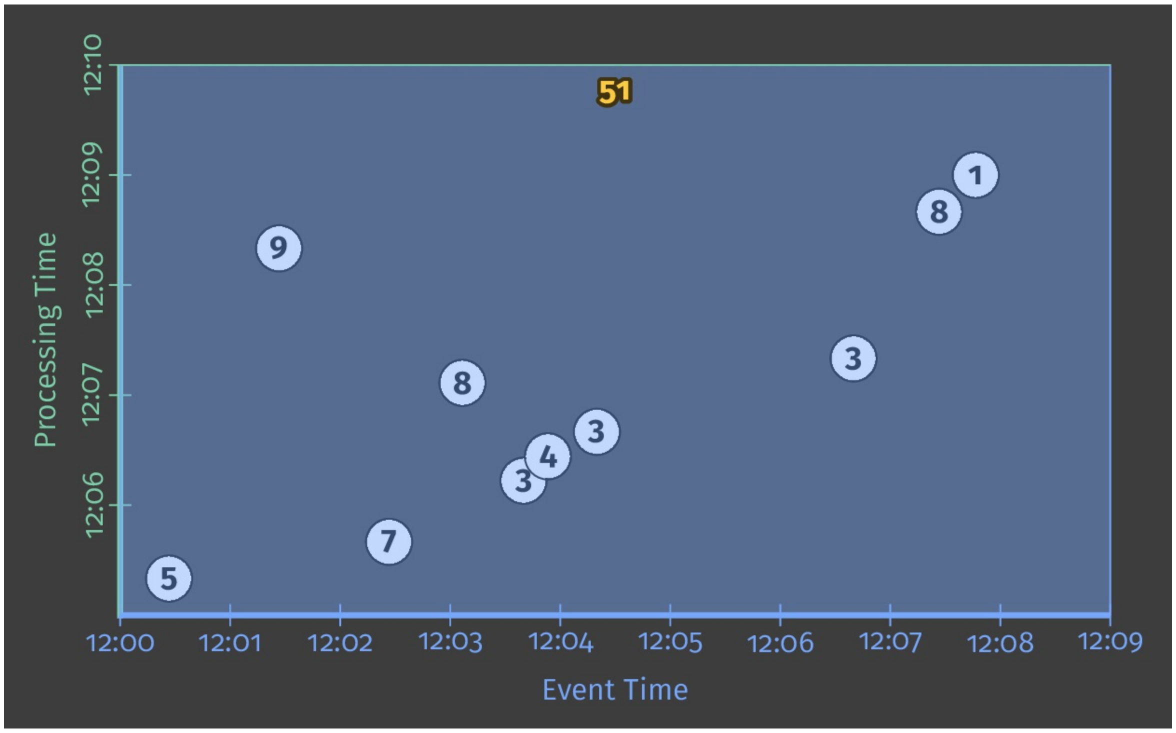

Figure 3-2. Classic batch processing.

Since 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 51. In this example, we're calculating a sum over all of event time, since 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, since it effectively never will. One of the concepts we'll want is windowing, which we introduced in Chapter 2. 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 the last chapter, windowing is the process of slicing up a data source along temporal boundaries. Common windowing strategies include fixed windows, sliding windows, and sessions windows:.

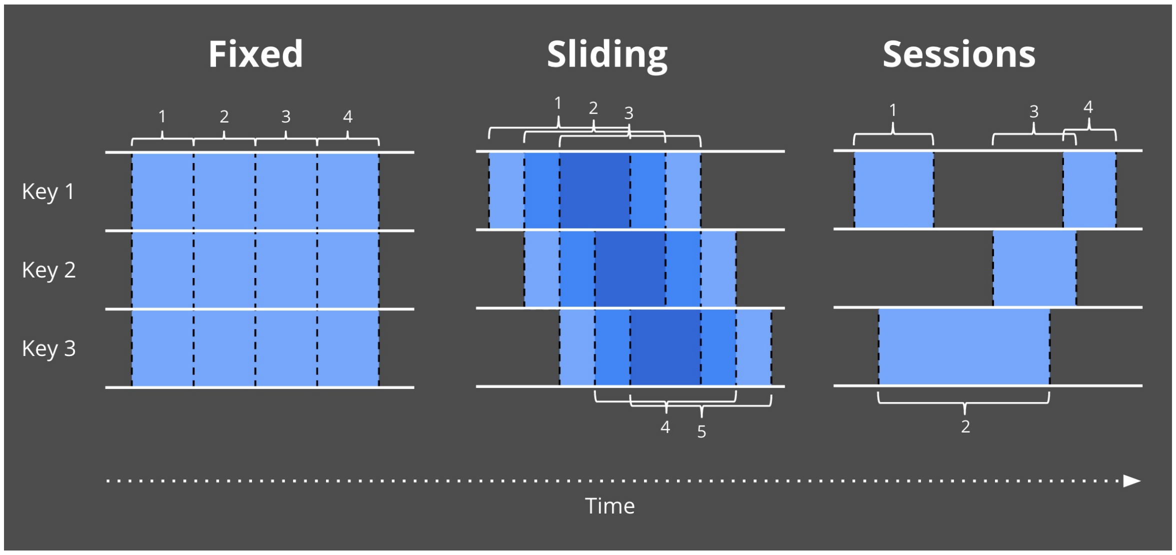

Figure 3-3. Example windowing strategies. Each example is shown for three different keys, highlighting the difference between aligned windows (which apply across all the data) and unaligned windows (which apply across a subset of the data).

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 (highlighted below):

Table 3-2. **Windowed summation code.**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
  .apply(Sum.integersPerKey());
```

Recall that Beam provides a unified model that works in both batch and streaming, since 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 dire

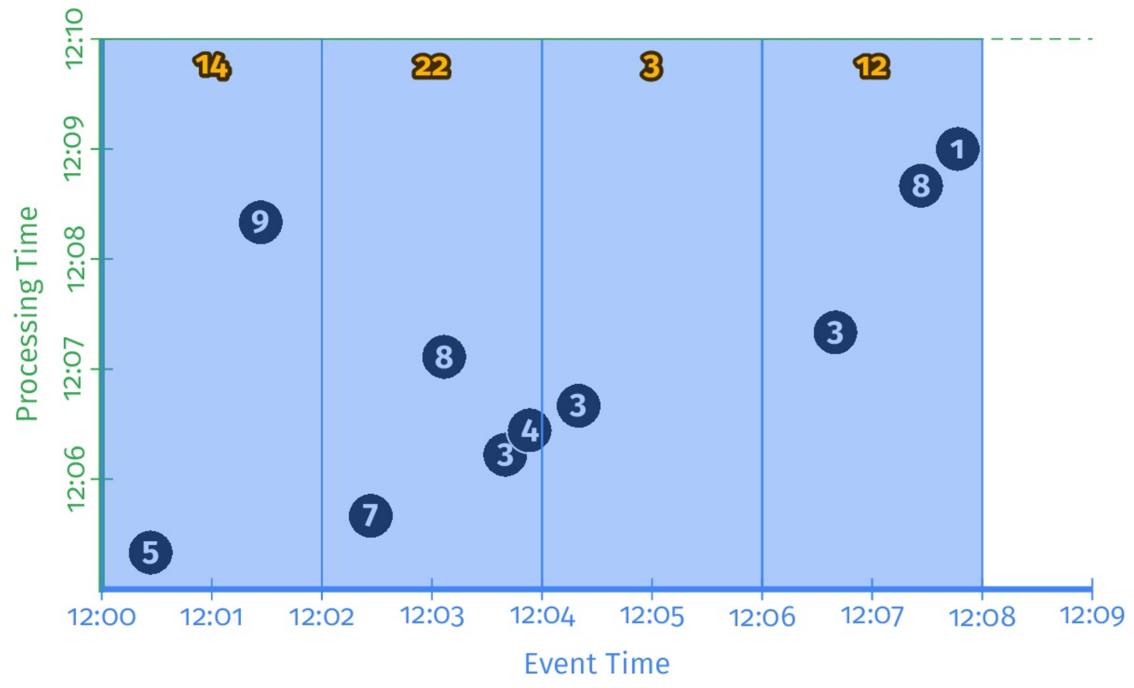

Figure 3-4. Windowed summation on a batch engine.

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 introduced in previous chapters: 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: watermarks, triggers, and accumulation.

Going Streaming: *When* & *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 whereas the batch engine had a known point at which the input for each window was complete (i.e., once all of the data in the bounded input source had been consumed), we currently lack a practical way of determining completeness with an unbounded data source. Enter watermarks.

When: Watermarks

Watermarks are the first half 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 here, where 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.

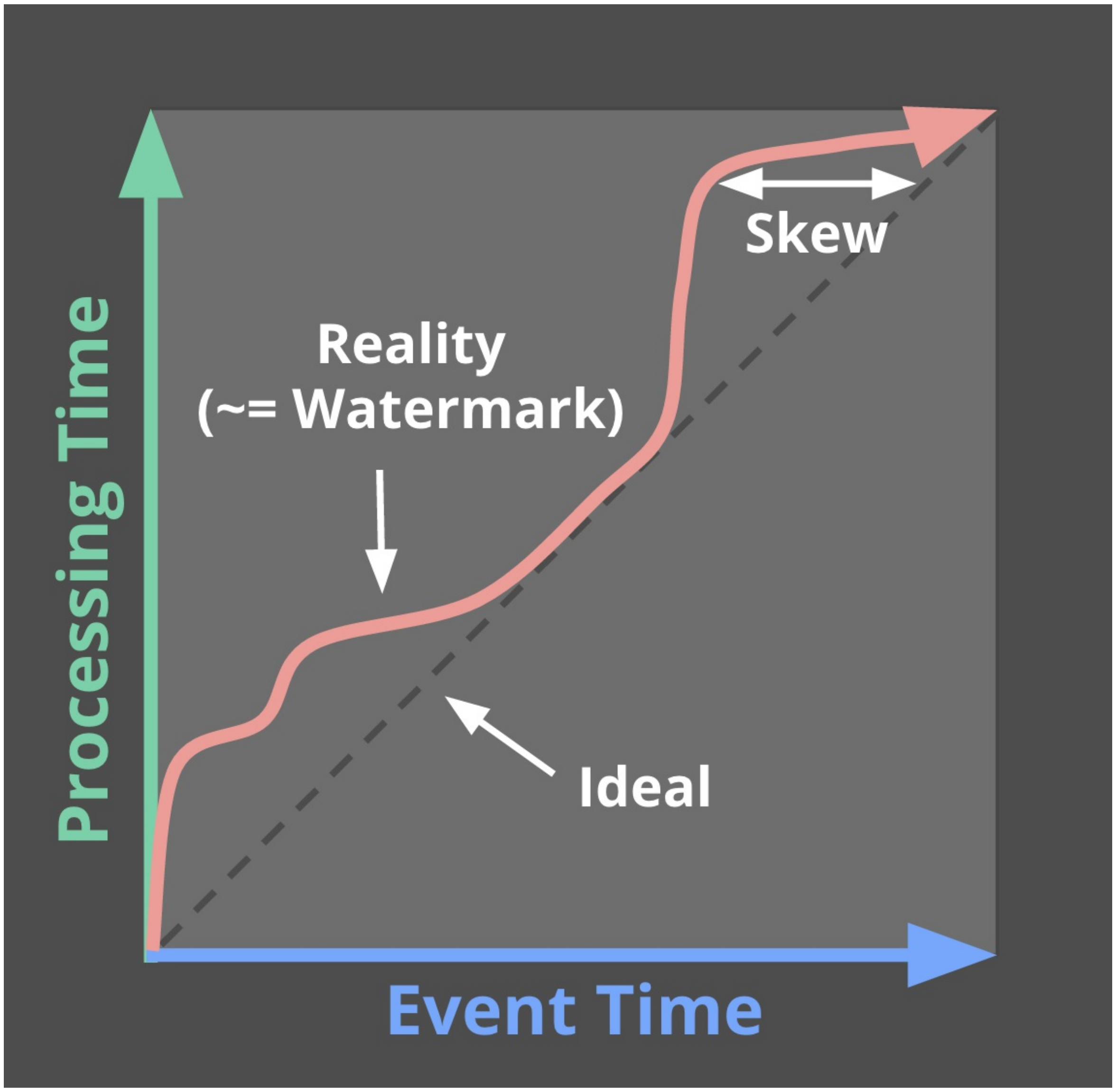

Figure 3-5. Event time progress, skew, and watermarks.

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) → E, which takes a point in processing time and returns a point in event time[3]. 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 may be a strict guarantee or an educated guess, respectively:

- **Perfect watermarks**: In the case where 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 it may sometimes be wrong, which will lead to late data. We’ll learn about ways to deal with late data in the triggers section below.

Watermarks are a fascinating and complex topic, as you’ll see when you get to Slava’s watermarks deep dive in Chapter 4. But for now, to get a better sense of the role that watermarks play, as well as some of their shortcomings, let’s look at two examples of a streaming engine using watermarks alone to determine when to materialize output while executing the windowed pipeline from Table 3-

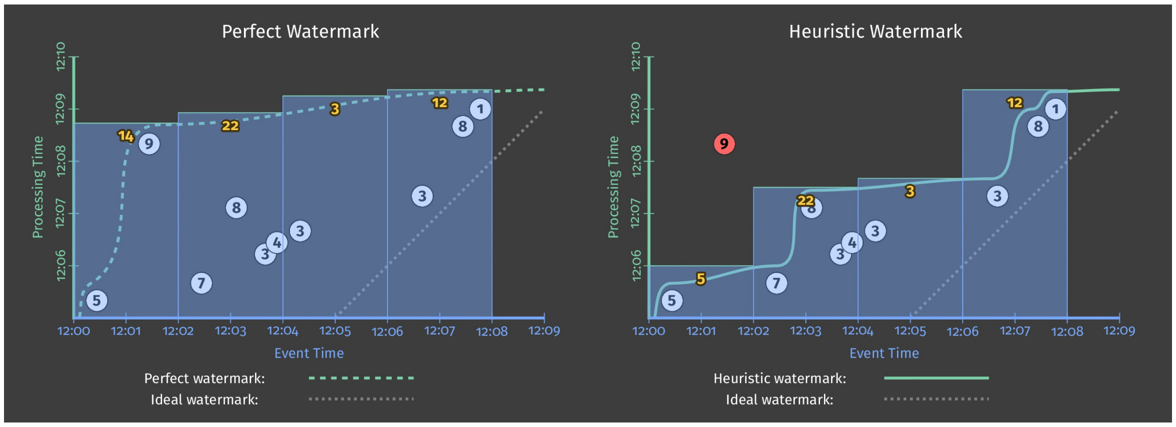

Figure 3-6. Windowed summation on a streaming engine with perfect (left) and heuristic (right) watermarks.

In both cases, windows are materialized as the watermark passes the end of the window. The primary difference between the two executions is that the heuristic algorithm used in watermark calculation on the right fails to take the value of 9 into account, which drastically changes the shape of the watermark[4]. These examples highlight two shortcomings of watermarks (and any other notion of completeness), specifically that they can be:

- **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, where 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), where 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: while 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’d 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 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. Addressing these shortcomings is where triggers come into play.

When: The wonderful thing about triggers, is triggers are wonderful things!

Triggers are the second half of 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 may make those decisions based off of things that happen in other time domains, such as watermarks progressing in the event time domain). Each specific output for a window is referred to as a pane of the window.

Examples of signals used for triggering include:

- **Watermark progress (i.e., event time progress)**, an implicit version of which we already saw in Figure 3.6, where outputs were materialized when the watermark passed the end of the window. Another use case is triggering garbage collection when the lifetime of a window exceeds some useful horizon, an example of which we'll see a little later on.

- **Processing time progress**, which is useful for providing regular, periodic updates, since processing time (unlike event time) always progresses more or less uniformly and without delay.

- **Element counts**, which are useful for triggering after some finite number of elements have been observed in a window.

- **Punctuations**, or other data-dependent triggers, where some record or feature of a record (e.g., an EOF element or a flush event) indicates that output should be generated.

In addition to simple triggers that fire based off of concrete signals, there are also composite triggers that allow for the creation of more sophisticated triggering logic. Example composite triggers include:

- **Repetitions**, which are particularly useful in conjunction with processing time triggers for providing regular, periodic updates.

- **Conjunctions** (logical AND), which fire only once all child triggers have fired (e.g., after the watermark passes the end of the window AND we observe a terminating punctuation record).

- **Disjunctions** (logical OR), which fire after any child triggers fire (e.g., after the watermark passes the end of the window OR we observe a terminating punctuation record).

- **Sequences**, which fire a progression of child triggers in a predefined order.

To make the notion of triggers a bit more concrete (and give us something to build upon), let's go ahead and make explicit the implicit default trigger used in Figure 3.6 by adding it to the code from Table 3-2:

Table 3-3. **Explicit default trigger.**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
               .triggering(AfterWatermark()))
  .apply(Sum.integersPerKey());
```

With that in mind, and a basic understanding of what triggers have to offer, we can look at tackling the problems of watermarks being too slow or too fast. In both cases, we essentially want to provide some sort of regular, materialized updates for a given window, either before or after the watermark advances past the end of the window (in addition to the update we'll receive at the threshold of the watermark passing the end of the window). So we'll want some sort of repetition trigger. The question then becomes: what are we repeating?

In the **too slow** case (i.e., providing early, speculative results), we probably should assume that there may be a steady amount of incoming data for any given window, since we know (by definition of being in the early stage for the window) that the input we've observed for the window is thus far incomplete. As such, triggering periodically when processing time advances (e.g., once per minute) is probably wise, because the number of trigger firings won't be dependent upon the amount of data actually observed for the window; at worst we'll just get a steady flow of periodic trigger firings.

In the **too fast** case (i.e., providing updated results in response to late data due to a heuristic watermark), let’s assume our watermark is based on a relatively accurate heuristic (often a reasonably safe assumption). In that case, we don’t expect to see late data very often, but when we do, it’d be nice to amend our results quickly. Triggering after observing an element count of one will give us quick updates to our results (i.e., immediately any time we see late data), but is not likely to overwhelm the system given the expected infrequency of late data.

Note that these are just examples: we’re free to choose different triggers (or to choose not to trigger at all for one or both of them) if appropriate for the use case at hand.

Lastly, we need to orchestrate the timing of these various triggers: early, on-time, and late. We can do this with a Sequence trigger and a special OrFinally trigger, which installs a child trigger that terminates the parent trigger when the child fires.

Table 3-4. **Manually-specified early and late firings.**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
               .triggering(Sequence(
                 Repeat(AtPeriod(Duration.standardMinutes(1)))
                   .OrFinally(AfterWatermark()),
                 Repeat(AfterCount(1))))
  .apply(Sum.integersPerKey());
```

However, that’s pretty wordy. And given that the pattern of repeated-early | on-time | repeated-late firings is so common, we provide a custom (but semantically equivalent) API in Beam to make specifying such triggers simpler and clearer:

Table 3-5. **Early and late firings via the early/late API.**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(
                     AtPeriod(Duration.standardMinutes(1)))
                   .withLateFirings(AtCount(1))))
  .apply(Sum.integersPerKey());
```

Executing either Table 3-4 or 3-5 on a streaming engine (with both perfect and heuristic watermarks, as before) then yields results that look like this:

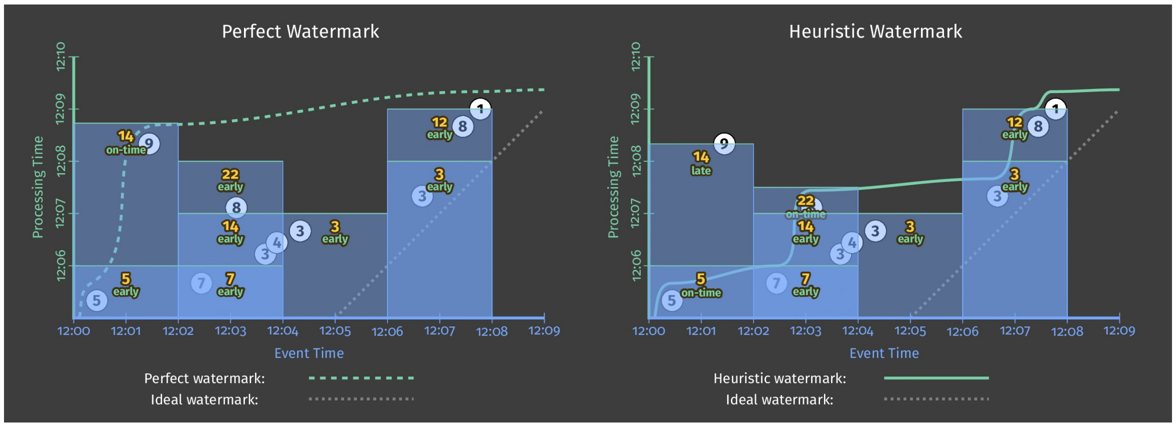

Figure 3-7. Windowed summation on a streaming engine with early and late firings.

This version has two clear improvements over Figure 3.6:

- 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, where 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, 14, then 22), 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 6 were starkly different, the two versions here look quite similar.

The biggest remaining difference at this point is window lifetime bounds. In the perfect watermark case, we know we’ll never see any more data for a window once 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 3.7, 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'd 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.

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, i.e., 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 arrive after this horizon are simply dropped. Once 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[5]. 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.

Since 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 Table 3-5/Figure 3.7 and add 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):

Table 3-6. **Early and late firings with allowed lateness**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(
                     AtPeriod(Duration.standardMinutes(1)))
                   .withLateFirings(AtCount(1)))
               .withAllowedLateness(Duration.standardMinutes(1)))
  .apply(Sum.integersPerKey());
```

The execution of this pipeline would look something like Figure 3.8 below, where I've added the following features to highlight the effects of allowed lateness:

- The thick white line denoting the current position in processing time is now annotated with ticks indicating the lateness horizon (in event time) for all active windows.

- Once the watermark passes the lateness horizon for a window, that window is closed, which means 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, so gets incorporated into an updated result with value 11. The 9, however, arrives beyond the lateness horizon, so is simply dropped.

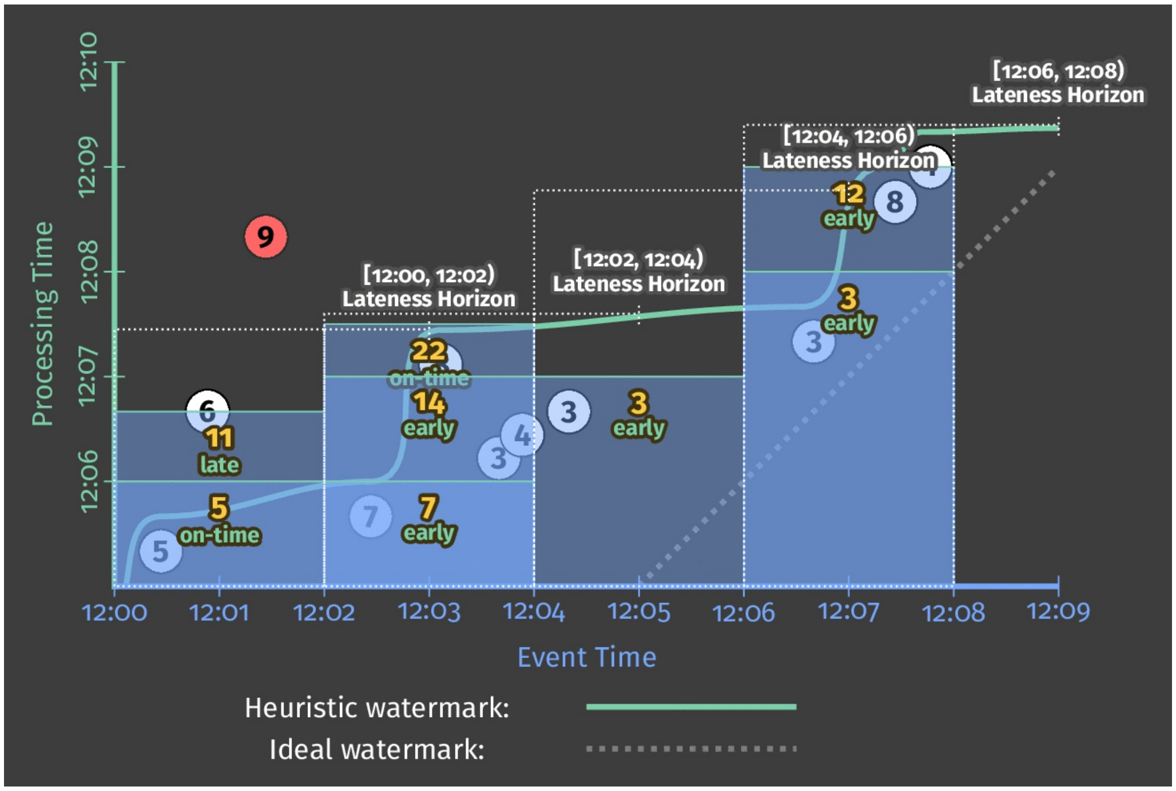

Figure 3-8. Windowed summation on a streaming engine with early and late firings and allowed lateness.

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, then 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 3.7.

- 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 built upon the one immediately preceding it. However, there are actually three different modes of accumulation[6]:

- **Discarding** - Every time a pane is materialized, any stored state is discarded. This means 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, e.g., when sending integers into a system that expects to receive deltas that it will sum together to produce a final count.

- **Accumulating** - As in Figure 3.7, every time a pane is materialized, any stored state is retained, and future inputs are accumulated into the existing state. This means 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 BigTable or HBase

- **Accumulating & Retracting** - Like accumulating mode, but when producing a new pane, 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 where retractions are particularly helpful:

    - When consumers downstream are **re-grouping 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’ll look at more closely below) are in use, the new value may be replacing more than one previous

The different semantics for each group are somewhat clearer when seen side-by-side. Consider the three panes for the second window in Figure 7 (the one with event time range [12:02, 12:04)). The table below shows what the values for each pane would look like across the three supported accumulation modes (with Accumulating mode being the specific mode used in Figure 3.7):

Table 3-7. Comparing accumulation modes using the second window from Fi

	Discarding	**Accumulating**	**Accumulating & Retr**
Pane 1: [7]	7	7	7
Pane 2: [3, 4]	7	14	14, -7
Pane 3: [8]	8	22	22, -14
Last Value Observed	8	22	22
Total Sum	22	51	22

- **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 the independent panes themselves, you would arrive at a correct answer of 22. 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 3.7, each pane incorporates the values that

arrived during that specific pane, plus all the values from previous panes. As such, the final value observed correctly captures the total sum of 22. If you were to sum up the individual panes themselves, however, you'd effectively be double- and triple-counting the inputs from panes 2 and 1, respectively, giving you an incorrect total sum of 51. This is why accumulating mode is most useful when you can simply overwrite previous values with new values: the new value already incorporates all the data seen thus far.

- **Accumulating & 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 (non-retraction) value observed as well as the total sum of all materialized panes (including retractions) provide you with the correct answer of 22. This is why retractions are so powerful.

To see discarding mode in action, we would make the following change to Table 3-5:

Table 3-8. **Discarding mode version of early/late firings.**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(
                     AtPeriod(Duration.standardMinutes(1)))
                   .withLateFirings(AtCount(1)))
               .discardingFiredPanes())
  .apply(Sum.integersPerKey());
```

Running again on a streaming engine with a heuristic watermark would produce output like the following:

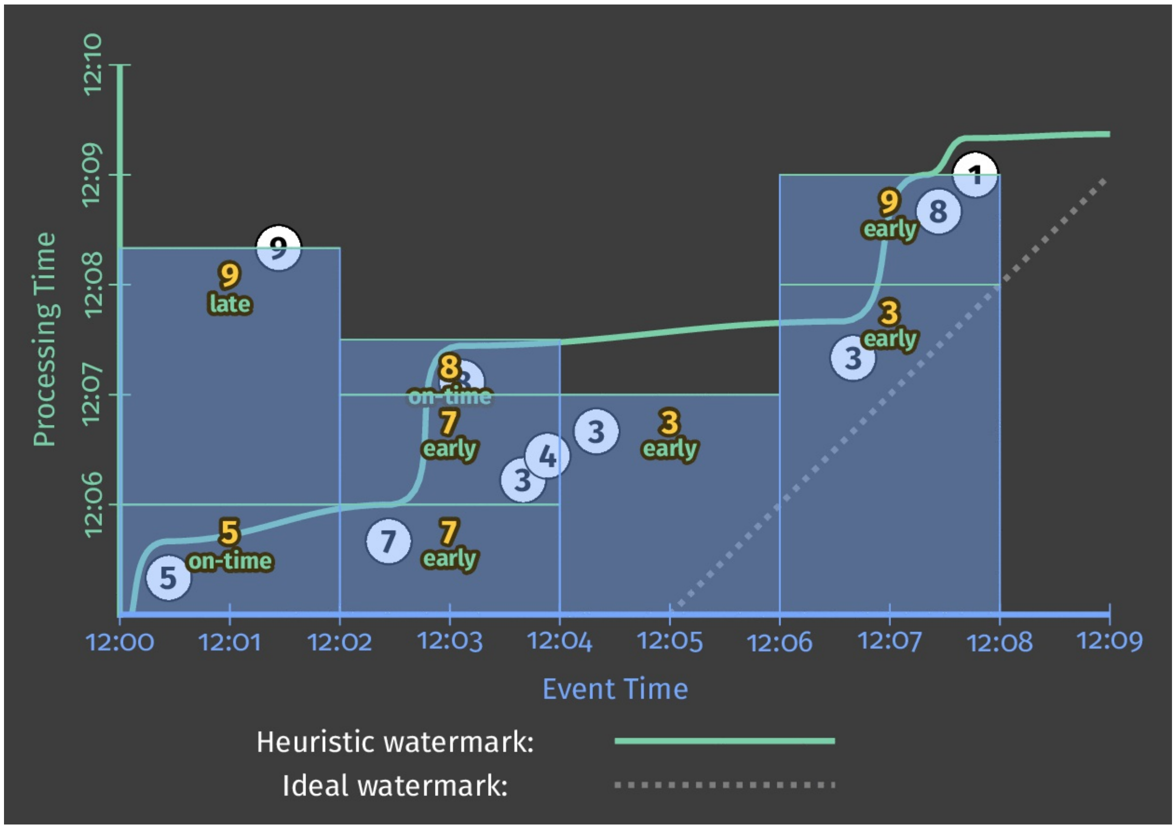

Figure 3-9. Discarding mode version of early/late firings on a streaming engine.

While the overall shape of the output is similar to the accumulating mode version from Figure 3.7, 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 (note, however, that despite my hopes otherwise, retractions remain unimplemented in Beam; if you want to help out, BEAM-91 is waiting patiently for you):

Table 3-9. **Accumulating & retracting mode version of early/late firi**

```
PCollection<KV<Team, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
               .triggering(
                 AfterWatermark()
                   .withEarlyFirings(
                     AtPeriod(Duration.standardMinutes(1)))
                   .withLateFirings(AtCount(1)))
               .accumulatingAndRetractingFiredPanes())
  .apply(Sum.integersPerKey());
```

And run on a streaming engine, this would yield output like the following:

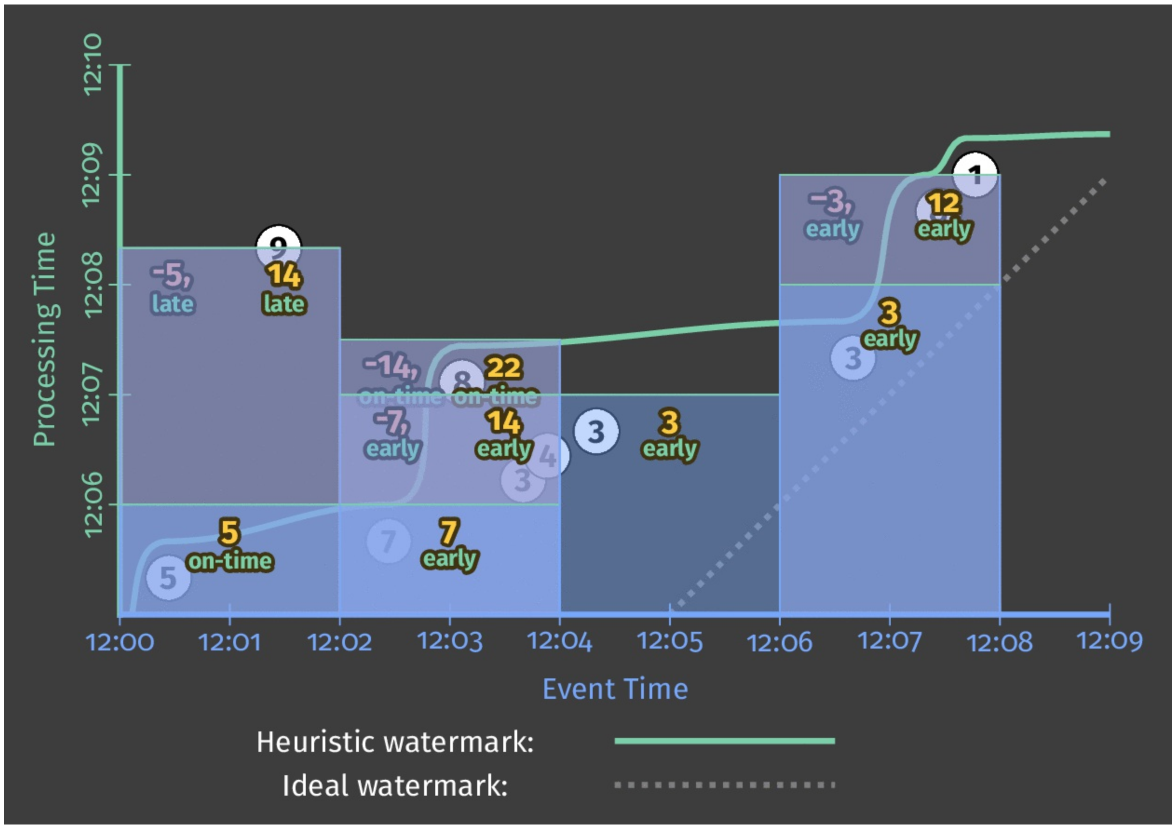

Figure 3-10. Accumulating & retracting mode version of early/late firings on a streaming engine.

Since 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.

Comparing the final frames of Figures 3.9, 3.7 (heuristic only), and 3.10 side-by-side provides a nice visual contrast of the three modes:

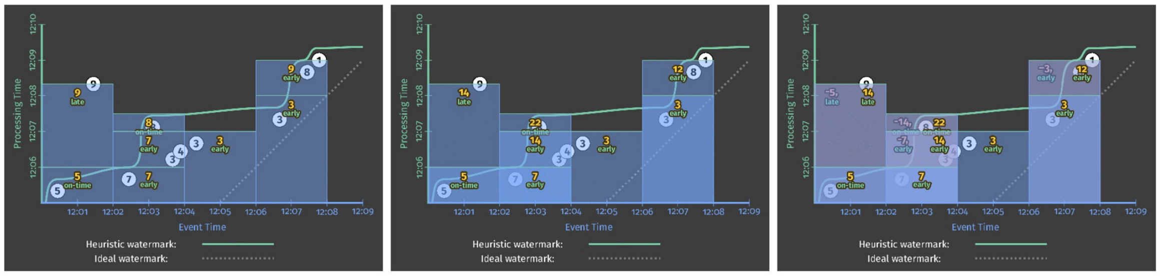

Figure 3-11. Side-by-side comparison of accumulation modes.

As you can imagine, the modes in the order presented (discarding, accumulating, accumulating & 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 tradeoffs along the axes of correctness, latency, and cost.

At this point, we've touched upon all four questions:

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

However, we've really only looked at one type of windowing: fixed windowing in event-time. As you know from Chapter 2, there are a number of dimensions to windowing, and I'd like to touch upon at least two of those before we call it day. First however, we're going to take a slight detour to dive further into the world of watermarks in Chapter 4, as this knowledge will help frame future discussions regarding some more advanced topics. Enter Slava, stage right.

[1] For those of you missing the animations from Streaming 102, there will be URLs to animated versions on the web as well; my attempts to convince O'Reilly to include flipbook versions of the animations in the appendix were regretfully unsuccessful.

[2] Bear with me here. Fine-grained emotional expressions via composite punctuation (i.e. emoticons) are strictly forbidden in O'Reilly publications.

[3] 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, etc.; but conceptually it’s simpler to think of it as a mapping from processing time to event time.

[4] 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, where 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. See chapter 4.

[5] It may 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.

[6] One might note that there should logically be a fourth mode, discarding and retracting; that mode isn’t terribly useful in most cases, so I won’t discuss it further here.