

Part I. Fundamentals of Stream Processing with Apache Spark

The first part of this book is dedicated to building solid foundations on the concepts that underpin stream processing and a theoretical understanding of Apache Spark as a streaming engine.

We begin with a discussion on what motivating drivers are behind the adoption of stream-processing techniques and systems in the enterprise today ([Chapter 1](#)). We then establish vocabulary and concepts common to stream processing ([Chapter 2](#)). Next, we take a quick look at how we got to the current state of the art as we discuss different streaming architectures ([Chapter 3](#)) and outline a theoretical understanding of Apache Spark as a streaming engine ([Chapter 4](#)).

At this point, the readers have the opportunity to directly jump to the more practical-oriented discussion of Structured Streaming in [Part II](#) or Spark Streaming in [Part III](#).

For those who prefer to gain a deeper understanding before adventuring into APIs and runtimes, we suggest that you continue reading about Spark's Distributed Processing model in [Chapter 5](#), in

which we lay the core concepts that will later help you to better understand the different implementations, options, and features offered by Spark Streaming and Structured Streaming.

In [Chapter 6](#), we deepen our understanding of the resilience model implemented by Spark and how it takes away the pain from the developer to implement robust streaming applications that can run enterprise-critical workloads 24/7.

With this new knowledge, we are ready to venture into the two streaming APIs of Spark, which we do in the subsequent parts of this book.

Chapter 1. Introducing Stream Processing

In 2011, Marc Andreessen famously said that “software is eating the world,” referring to the booming digital economy, at a time when many enterprises were facing the challenges of a digital transformation. Successful online businesses, using “online” and “mobile” operation modes, were taking over their traditional “brick-and-mortar” counterparts.

For example, imagine the traditional experience of buying a new camera in a photography shop: we would visit the shop, browse around, maybe ask a few questions of the clerk, make up our mind, and finally buy a model that fulfilled our desires and expectations. After finishing our purchase, the shop would have registered a credit card transaction—or only a cash balance change in case of a cash payment—and the shop manager would know they have one less inventory item of that particular camera model.

Now, let’s take that experience online: first, we begin searching the web. We visit a couple of online stores, leaving digital traces as we pass from one to another. Advertisements on websites suddenly begin showing us promotions for the camera we were looking at as well as for competing alternatives. We finally find an online shop offering us the best deal and purchase the camera. We create an account. Our personal data is registered and linked to the purchase. While we

complete our purchase, we are offered additional options that are allegedly popular with other people who bought the same camera. Each of our digital interactions, like searching for keywords on the web, clicking some link, or spending time reading a particular page generates a series of events that are collected and transformed into business value, like personalized advertisement or upsale recommendations.

Commenting on Andreessen's quote, in 2015, Dries Buytaert said "no, actually, *data* is eating the world." What he meant is that the disruptive companies of today are no longer disruptive because of their software, but because of the unique data they collect and their ability to transform that data into value.

The adoption of stream-processing technologies is driven by the increasing need of businesses to improve the time required to react and adapt to changes in their operational environment. This way of processing data as it comes in provides a technical and strategical advantage. Examples of this ongoing adoption include sectors such as internet commerce, continuously running data pipelines created by businesses that interact with customers on a 24/7 basis, or credit card companies, analyzing transactions as they happen in order to detect and stop fraudulent activities as they happen.

Another driver of stream processing is that our ability to generate data far surpasses our ability to make sense of it. We are constantly increasing the number of computing-capable devices in our personal and professional environments<m-dash>televisions, connected cars, smartphones, bike computers, smart watches, surveillance cameras,

thermostats, and so on. We are surrounding ourselves with devices meant to produce event logs: streams of messages representing the actions and incidents that form part of the history of the device in its context. As we interconnect those devices more and more, we create an ability for us to access and therefore analyze those event logs. This phenomenon opens the door to an incredible burst of creativity and innovation in the domain of near real-time data analytics, on the condition that we find a way to make this analysis tractable. In this world of aggregated event logs, stream processing offers the most resource-friendly way to facilitate the analysis of streams of data.

It is not a surprise that not only is data eating the world, but so is *streaming data*.

In this chapter, we start our journey in stream processing using Apache Spark. To prepare us to discuss the capabilities of Spark in the stream-processing area, we need to establish a common understanding of what stream processing is, its applications, and its challenges. After we build that common language, we introduce Apache Spark as a generic data-processing framework able to handle the requirements of batch and streaming workloads using a unified model. Finally, we zoom in on the streaming capabilities of Spark, where we present the two available APIs: Spark Streaming and Structured Streaming. We briefly discuss their salient characteristics to provide a sneak peek into what you will discover in the rest of this book.

What Is Stream Processing?

Stream processing is the discipline and related set of techniques used to extract information from *unbounded data*.

In his book *Streaming Systems*, Tyler Akidau defines unbounded data as follows:

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

Given that our information systems are built on hardware with finite resources such as memory and storage capacity, they cannot possibly hold unbounded datasets. Instead, we observe the data as it is received at the processing system in the form of a flow of events over time. We call this a *stream* of data.

In contrast, we consider *bounded data* as a dataset of known size. We can count the number of elements in a bounded dataset.

Batch Versus Stream Processing

How do we process both types of datasets? With *batch processing*, we refer to the computational analysis of bounded datasets. In practical terms, this means that those datasets are available and retrievable as a whole from some form of storage. We know the size of the dataset at the start of the computational process, and the duration of that process is limited in time.

In contrast, with *stream processing* we are concerned with the processing of data as it arrives to the system. Given the unbounded nature of data streams, the stream processors need to run constantly

for as long as the stream is delivering new data. That, as we learned, might be—theoretically—forever.

Stream-processing systems apply programming and operational techniques to make possible the processing of potentially infinite data streams with a limited amount of computing resources.

The Notion of Time in Stream Processing

Data can be encountered in two forms:

- At rest, in the form of a file, the contents of a database, or some other kind of record
- In motion, as continuously generated sequences of signals, like the measurement of a sensor or GPS signals from moving vehicles

We discussed already that a stream-processing program is a program that assumes its input is potentially infinite in size. More specifically, a stream-processing program assumes that its input is a sequence of signals of indefinite length, *observed over time*.

From the point of view of a timeline, *data at rest* is data from the past: arguably, all bounded datasets, whether stored in files or contained in databases, were initially a stream of data collected over time into some storage. The user’s database, all the orders from the last quarter, the GPS coordinates of taxi trips in a city, and so on all started as individual events collected in a repository.

Trying to reason about *data in motion* is more challenging. There is a time difference between the moment data is originally generated and when it becomes available for processing. That time delta might be very short, like web log events generated and processed within the same datacenter, or much longer, like GPS data of a car traveling through a tunnel that is dispatched only when the vehicle reestablishes its wireless connectivity after it leaves the tunnel.

We can observe that there's a timeline when the events were produced and another for when the events are handled by the stream-processing system. These timelines are so significant that we give them specific names:

Event time

The time when the event was created. The time information is provided by the local clock of the device generating the event.

Processing time

The time when the event is handled by the stream-processing system. This is the clock of the server running the processing logic. It's usually relevant for technical reasons like computing the processing lag or as criteria to determine duplicated output.

The differentiation among these timelines becomes very important when we need to correlate, order, or aggregate the events with respect to one another.

The Factor of Uncertainty

In a timeline, data at rest relates to the past, and data in motion can be seen as the present. But what about the future? One of the most subtle

aspects of this discussion is that it makes no assumptions on the throughput at which the system receives events.

In general, streaming systems do not require the input to be produced at regular intervals, all at once, or following a certain rhythm. This means that, because computation usually has a cost, it's a challenge to predict peak load: matching the sudden arrival of input elements with the computing resources necessary to process them.

If we have the computing capacity needed to match a sudden influx of input elements, our system will produce results as expected, but if we have not planned for such a burst of input data, some streaming systems might face delays, resource constriction, or failure.

Dealing with uncertainty is an important aspect of stream processing.

In summary, stream processing lets us extract information from infinite data streams observed as events delivered over time.

Nevertheless, as we receive and process data, we need to deal with the additional complexity of event-time and the uncertainty introduced by an unbounded input.

Why would we want to deal with the additional trouble? In the next section, we glance over a number of use cases that illustrate the value added by stream processing and how it delivers on the promise of providing faster, actionable insights, and hence business value, on data streams.

Some Examples of Stream Processing

The use of stream processing goes as wild as our capacity to imagine new real-time, innovative applications of data. The following use cases, in which the authors have been involved in one way or another, are only a small sample that we use to illustrate the wide spectrum of application of stream processing:

Device monitoring

A small startup rolled out a cloud-based Internet of Things (IoT) device monitor able to collect, process, and store data from up to 10 million devices. Multiple stream processors were deployed to power different parts of the application, from real-time dashboard updates using in-memory stores, to continuous data aggregates, like unique counts and minimum/maximum measurements.

Fault detection

A large hardware manufacturer applies a complex stream-processing pipeline to receive device metrics. Using time-series analysis, potential failures are detected and corrective measures are automatically sent back to the device.

Billing modernization

A well-established insurance company moved its billing system to a streaming pipeline. Batch exports from its existing mainframe infrastructure are streamed through this system to meet the existing billing processes while allowing new real-time flows from insurance agents to be served by the same logic.

Fleet management

A fleet management company installed devices able to report real-time data from the managed vehicles, such as location, motor parameters, and fuel levels, allowing it to enforce rules like

geographical limits and analyze driver behavior regarding speed limits.

Media recommendations

A national media company deployed a streaming pipeline to ingest new videos, such as news reports, into its recommendation system, making the videos available to its users' personalized suggestions almost as soon as they are ingested into the company's media repository. The company's previous system would take hours to do the same.

Faster loans

A bank active in loan services was able to reduce loan approval from hours to seconds by combining several data streams into a streaming application.

A common thread among those use cases is the need of the business to process the data and create actionable insights in a short period of time from when the data was received. This time is relative to the use case: although *minutes* is a very fast turn-around for a loan approval, a milliseconds response is probably necessary to detect a device failure and issue a corrective action within a given service-level threshold.

In all cases, we can argue that *data* is better when consumed as fresh as possible.

Now that we have an understanding of what stream processing is and some examples of how it is being used today, it's time to delve into the concepts that underpin its implementation.

Scaling Up Data Processing

Before we discuss the implications of distributed computation in stream processing, let's take a quick tour through *MapReduce*, a computing model that laid the foundations for scalable and reliable data processing.

MapReduce

The history of programming for distributed systems experienced a notable event in February 2003. Jeff Dean and Sanjay Gemawhat, after going through a couple of iterations of rewriting Google's crawling and indexing systems, began noticing some operations that they could expose through a common interface. This led them to develop *MapReduce*, a system for distributed processing on large clusters at Google.

Part of the reason we didn't develop MapReduce earlier was probably because when we were operating at a smaller scale, then our computations were using fewer machines, and therefore robustness wasn't quite such a big deal: it was fine to periodically checkpoint some computations and just restart the whole computation from a checkpoint if a machine died. Once you reach a certain scale, though, that becomes fairly untenable since you'd always be restarting things and never make any forward progress.

—Jeff Dean, email to Bradford F. Lyon, August 2013

MapReduce is a programming API first, and a set of components second, that make programming for a distributed system a relatively easier task than all of its predecessors.

Its core tenets are two functions:

Map

The map operation takes as an argument a function to be applied to every element of the collection. The collection's elements are read in a distributed manner, through the distributed filesystem, one chunk per executor machine. Then, all of the elements of the collection that reside in the local chunk see the function applied to them, and the executor emits the result of that application, if any.

Reduce

The reduce operation takes two arguments: one is a neutral element, which is what the *reduce* operation would return if passed an empty collection. The other is an aggregation operation, that takes the current value of an aggregate, a new element of the collection, and lumps them into a new aggregate.

Combinations of these two higher-order functions are powerful enough to express every operation that we would want to do on a dataset.

The Lesson Learned: Scalability and Fault Tolerance

From the programmer's perspective, here are the main advantages of MapReduce:

- It has a simple API.
- It offers very high expressivity.
- It significantly offloads the difficulty of distributing a program from the shoulders of the programmer to those of the library designer. In particular, resilience is built into the model.

Although these characteristics make the model attractive, the main success of MapReduce is its ability to sustain growth. As data volumes increase and growing business requirements lead to more information-extraction jobs, the MapReduce model demonstrates two crucial properties:

Scalability

As datasets grow, it is possible to add more resources to the cluster of machines in order to preserve a stable processing performance.

Fault tolerance

The system can sustain and recover from partial failures. All data is replicated. If a data-carrying executor crashes, it is enough to relaunch the task that was running on the crashed executor. Because the master keeps track of that task, that does not pose any particular problem other than rescheduling.

These two characteristics combined result in a system able to constantly sustain workloads in an environment fundamentally unreliable, *properties that we also require for stream processing*.

Distributed Stream Processing

One fundamental difference of stream processing with the MapReduce model, and with batch processing in general, is that although batch processing has access to the complete dataset, with streams, we see only a small portion of the dataset at any time.

This situation becomes aggravated in a distributed system; that is, in an effort to distribute the processing load among a series of executors, we further split up the input stream into partitions. Each executor gets to see only a partial view of the complete stream.

The challenge for a distributed stream-processing framework is to provide an abstraction that hides this complexity from the user and lets us reason about the stream as a whole.

Stateful Stream Processing in a Distributed System

Let's imagine that we are counting the votes during a presidential election. The classic batch approach would be to wait until all votes have been cast and then proceed to count them. Even though this approach produces a correct end result, it would make for very boring news over the day because no (intermediate) results are known until the end of the electoral process.

A more exciting scenario is when we can count the votes per candidate as each vote is cast. At any moment, we have a partial count by participant that lets us see the current standing as well as the voting trend. We can probably anticipate a result.

To accomplish this scenario, the stream processor needs to keep an internal register of the votes seen so far. To ensure a consistent count, this register must recover from any partial failure. Indeed, we can't ask the citizens to issue their vote again due to a technical failure.

Also, any eventual failure recovery cannot affect the final result. We can't risk declaring the wrong winning candidate as a side effect of an

ill-recovered system.

This scenario illustrates the challenges of stateful stream processing running in a distributed environment. Stateful processing poses additional burdens on the system:

- We need to ensure that the state is preserved over time.
- We require data consistency guarantees, even in the event of partial system failures.

As you will see throughout the course of this book, addressing these concerns is an important aspect of stream processing.

Now that we have a better sense of the drivers behind the popularity of stream processing and the challenging aspects of this discipline, we can introduce Apache Spark. As a unified data analytics engine, Spark offers data-processing capabilities for both batch and streaming, making it an excellent choice to satisfy the demands of the data-intensive applications, as we see next.

Introducing Apache Spark

Apache Spark is a fast, reliable, and fault-tolerant distributed computing framework for large-scale data processing.

The First Wave: Functional APIs

In its early days, Spark's breakthrough was driven by its novel use of memory and expressive functional API. The Spark memory model uses RAM to cache data as it is being processed, resulting in up to

100 times faster processing than Hadoop MapReduce, the open source implementation of Google’s MapReduce for batch workloads.

Its core abstraction, the *Resilient Distributed Dataset* (RDD), brought a rich functional programming model that abstracted out the complexities of distributed computing on a cluster. It introduced the concepts of *transformations* and *actions* that offered a more expressive programming model than the map and reduce stages that we discussed in the MapReduce overview. In that model, many available *transformations* like `map`, `flatmap`, `join`, and `filter` express the lazy conversion of the data from one internal representation to another, whereas eager operations called *actions* materialize the computation on the distributed system to produce a result.

The Second Wave: SQL

The second game-changer in the history of the Spark project was the introduction of Spark SQL and *DataFrames* (and later, *Dataset*, a strongly typed DataFrame). From a high-level perspective, Spark SQL adds SQL support to any dataset that has a schema. It makes it possible to query a comma-separated values (CSV), Parquet, or JSON dataset in the same way that we used to query a SQL database.

This evolution also lowered the threshold of adoption for users. Advanced distributed data analytics were no longer the exclusive realm of software engineers; it was now accessible to data scientists, business analysts, and other professionals familiar with SQL. From a performance point of view, SparkSQL brought a query optimizer and

a physical execution engine to Spark, making it even faster while using fewer resources.

A Unified Engine

Nowadays, Spark is a unified analytics engine offering batch and streaming capabilities that is compatible with a polyglot approach to data analytics, offering APIs in Scala, Java, Python, and the R language.

While in the context of this book we are going to focus our interest on the streaming capabilities of Apache Spark, its batch functionality is equally advanced and is highly complementary to streaming applications. Spark's unified programming model means that developers need to learn only one new paradigm to address both batch and streaming workloads.

NOTE

In the course of the book, we use *Apache Spark* and *Spark* interchangeably. We use *Apache Spark* when we want to make emphasis on the project or open source aspect of it, whereas we use *Spark* to refer to the technology in general.

Spark Components

Figure 1-1 illustrates how Spark consists of a core engine, a set of abstractions built on top of it (represented as horizontal layers), and libraries that use those abstractions to address a particular area (vertical boxes). We have highlighted the areas that are within the scope of this book and grayed out those that are not covered. To learn

more about these other areas of Apache Spark, we recommend *Spark, The Definitive Guide* by Bill Chambers and Matei Zaharia (O'Reilly), and *High Performance Spark* by Holden Karau and Rachel Warren (O'Reilly).

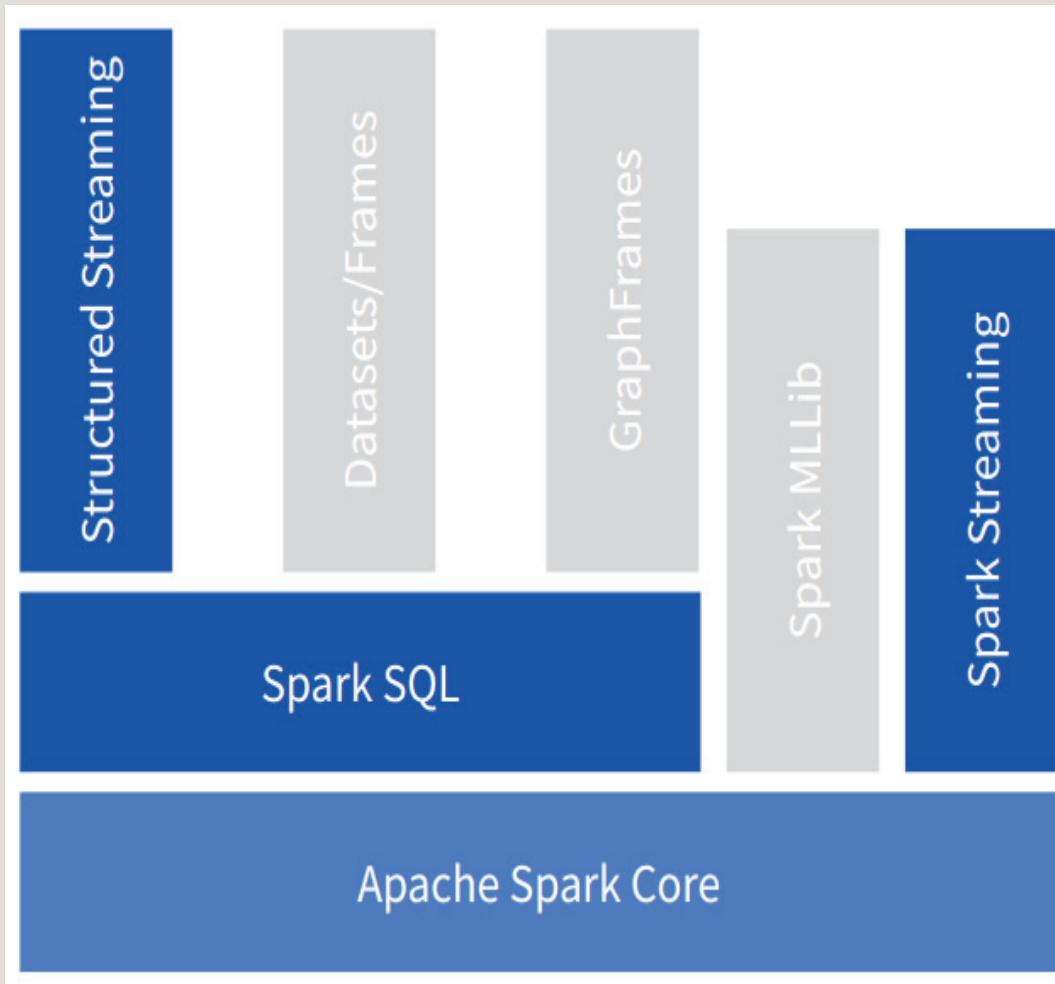


Figure 1-1. Abstraction layers (horizontal) and libraries (vertical) offered by Spark

As abstraction layers in Spark, we have the following:

Spark Core

Contains the Spark core execution engine and a set of low-level functional APIs used to distribute computations to a cluster of computing resources, called *executors* in Spark lingo. Its cluster

abstraction allows it to submit workloads to YARN, Mesos, and Kubernetes, as well as use its own standalone cluster mode, in which Spark runs as a dedicated service in a cluster of machines. Its datasource abstraction enables the integration of many different data providers, such as files, block stores, databases, and event brokers.

Spark SQL

Implements the higher-level `Dataset` and `DataFrame` APIs of Spark and adds SQL support on top of arbitrary data sources. It also introduces a series of performance improvements through the Catalyst query engine, and code generation and memory management from project Tungsten.

The libraries built on top of these abstractions address different areas of large-scale data analytics: *MLlib* for machine learning, *GraphFrames* for graph analysis, and the two APIs for stream processing that are the focus of this book: Spark Streaming and Structured Streaming.

Spark Streaming

Spark Streaming was the first stream-processing framework built on top of the distributed processing capabilities of the core Spark engine. It was introduced in the Spark 0.7.0 release in February of 2013 as an alpha release that evolved over time to become today a mature API that's widely adopted in the industry to process large-scale data streams.

Spark Streaming is conceptually built on a simple yet powerful premise: apply Spark's distributed computing capabilities to stream processing by transforming continuous streams of data into discrete

data collections on which Spark could operate. This approach to stream processing is called the *microbatch* model; this is in contrast with the *element-at-time* model that dominates in most other stream-processing implementations.

Spark Streaming uses the same functional programming paradigm as the Spark core, but it introduces a new abstraction, the *Discretized Stream* or *DStream*, which exposes a programming model to operate on the underlying data in the stream.

Structured Streaming

Structured Streaming is a stream processor built on top of the Spark SQL abstraction. It extends the Dataset and DataFrame APIs with streaming capabilities. As such, it adopts the schema-oriented transformation model, which confers the *structured* part of its name, and inherits all the optimizations implemented in Spark SQL.

Structured Streaming was introduced as an experimental API with Spark 2.0 in July of 2016. A year later, it reached *general availability* with the Spark 2.2 release becoming eligible for production deployments. As a relatively new development, Structured Streaming is still evolving fast with each new version of Spark.

Structured Streaming uses a declarative model to acquire data from a stream or set of streams. To use the API to its full extent, it requires the specification of a schema for the data in the stream. In addition to supporting the general transformation model provided by the Dataset and DataFrame APIs, it introduces stream-specific

features such as support for event-time, streaming joins, and separation from the underlying runtime. That last feature opens the door for the implementation of runtimes with different execution models. The default implementation uses the classical microbatch approach, whereas a more recent *continuous processing* backend brings experimental support for near-real-time continuous execution mode.

Structured Streaming delivers a unified model that brings stream processing to the same level of batch-oriented applications, removing a lot of the cognitive burden of reasoning about stream processing.

Where Next?

If you are feeling the urge to learn either of these two APIs right away, you could directly jump to Structured Streaming in [Part II](#) or Spark Streaming in [Part III](#).

If you are not familiar with stream processing, we recommend that you continue through this initial part of the book because we build the vocabulary and common concepts that we use in the discussion of the specific frameworks.

Chapter 2. Stream-Processing Model

In this chapter, we bridge the notion of a data stream—a source of data “on the move”—with the programming language primitives and constructs that allow us to express stream processing.

We want to describe simple, fundamental concepts first before moving on to how Apache Spark represents them. Specifically, we want to cover the following as components of stream processing:

- Data sources
- Stream-processing pipelines
- Data sinks

We then show how those concepts map to the specific stream-processing model implemented by Apache Spark.

Next, we characterize stateful stream processing, a type of stream processing that requires bookkeeping of past computations in the form of some intermediate state needed to process new data. Finally, we consider streams of timestamped events and basic notions involved in addressing concerns such as “what do I do if the order and timeliness of the arrival of those events do not match expectations?”

Sources and Sinks

As we mentioned earlier, Apache Spark, in each of its two streaming systems—Structured Streaming and Spark Streaming—is a programming framework with APIs in the Scala, Java, Python, and R programming languages. It can only operate on data that enters the runtime of programs using this framework, and it ceases to operate on the data as soon as it is being sent to another system.

This is a concept that you are probably already familiar with in the context of data at rest: to operate on data stored as a file of records, we need to read that file into memory where we can manipulate it, and as soon as we have produced an output by computing on this data, we have the ability to write that result to another file. The same principle applies to databases—another example of data at rest.

Similarly, data streams can be made accessible as such, in the streaming framework of Apache Spark using the concept of streaming *data sources*. In the context of stream processing, accessing data from a stream is often referred to as *consuming the stream*. This abstraction is presented as an interface that allows the implementation of instances aimed to connect to specific systems: Apache Kafka, Flume, Twitter, a TCP socket, and so on.

Likewise, we call the abstraction used to write a data stream outside of Apache Spark’s control a *streaming sink*. Many connectors to various specific systems are provided by the Spark project itself as well as by a rich ecosystem of open source and commercial third-party integrations.

In Figure 2-1, we illustrate this concept of sources and sinks in a stream-processing system. Data is consumed from a source by a processing component and the eventual results are produced to a sink.



Figure 2-1. Simplified streaming model

The notion of sources and sinks represents the system's boundary. This labeling of system boundaries makes sense given that a distributed framework can have a highly complex footprint among our computing resources. For example, it is possible to connect an Apache Spark cluster to another Apache Spark cluster, or to another distributed system, of which Apache Kafka is a frequent example. In that context, one framework's sink is the downstream framework's source. This chaining is commonly known as a *pipeline*. The name of sources and sinks is useful to both describe data passing from one system to the next and which point of view we are adopting when speaking about each system independently.

Immutable Streams Defined from One Another

Between sources and sinks lie the programmable constructs of a stream-processing framework. We do not want to get into the details of this subject yet—you will see them appear later in Part II and Part III for Structured Streaming and Spark Streaming, respectively.

But we can introduce a few notions that will be useful to understand how we express stream processing.

Both stream APIs in Apache Spark take the approach of functional programming: they declare the transformations and aggregations they operate on data streams, assuming that those streams are immutable. As such, for one given stream, it is impossible to mutate one or several of its elements. Instead, we use transformations to express how to process the contents of one stream to obtain a derived data stream. This makes sure that at any given point in a program, any data stream can be traced to its inputs by a sequence of transformations and operations that are explicitly declared in the program. As a consequence, any particular process in a Spark cluster can reconstitute the content of the data stream using only the program and the input data, making computation unambiguous and reproducible.

Transformations and Aggregations

Spark makes extensive use of *transformations* and *aggregations*. Transformations are computations that express themselves in the same way for every element in the stream. For example, creating a derived stream that doubles every element of its input stream corresponds to a transformation. Aggregations, on the other hand, produce results that depend on many elements and potentially every element of the stream observed until now. For example, collecting the top five largest numbers of an input stream corresponds to an aggregation. Computing the average value of some reading every 10 minutes is also an example of an aggregation.

Another way to designate those notions is to say that transformations have *narrow dependencies* (to produce one element of the output, you need only one of the elements of the input), whereas aggregations have *wide dependencies* (to produce one element of the output you would need to observe many elements of the input stream encountered so far). This distinction is useful. It lets us envision a way to express basic functions that produces results using higher-order functions.

Although Spark Streaming and Structured Streaming have distinct ways of representing a data stream, the APIs they operate on are similar in nature. They both present themselves under the form of a series of transformations applied to immutable input streams and produce an output stream, either as a bona fide data stream or as an output operation (data sink).

Window Aggregations

Stream-processing systems often feed themselves on actions that occur in real time: social media messages, clicks on web pages, ecommerce transactions, financial events, or sensor readings are also frequently encountered examples of such events. Our streaming application often has a centralized view of the logs of several places, whether those are retail locations or simply web servers for a common application. Even though seeing every transaction individually might not be useful or even practical, we might be interested in seeing the properties of events seen over a recent period of time; for example, the last 15 minutes or the last hour, or maybe even both.

Moreover, the very idea of stream processing is that the system is supposed to be long-running, dealing with a continuous stream of data. As these events keep coming in, the older ones usually become less and less relevant to whichever processing you are trying to accomplish.

We find many applications of regular and recurrent time-based aggregations that we call *windows*.

Tumbling Windows

The most natural notion of a window aggregation is that of “a grouping function each x period of time.” For instance, “the maximum and minimum ambient temperature each hour” or “the total energy consumption (kW) each 15 minutes” are examples of window aggregations. Notice how the time periods are inherently consecutive and nonoverlapping. We call this grouping of a fixed time period, in which each group follows the previous and does not overlap, *tumbling windows*.

Tumbling windows are the norm when we need to produce aggregates of our data over regular periods of time, with each period independent from previous periods. Figure 2-2 shows a tumbling window of 10 seconds over a stream of elements. This illustration demonstrates the tumbling nature of tumbling windows.

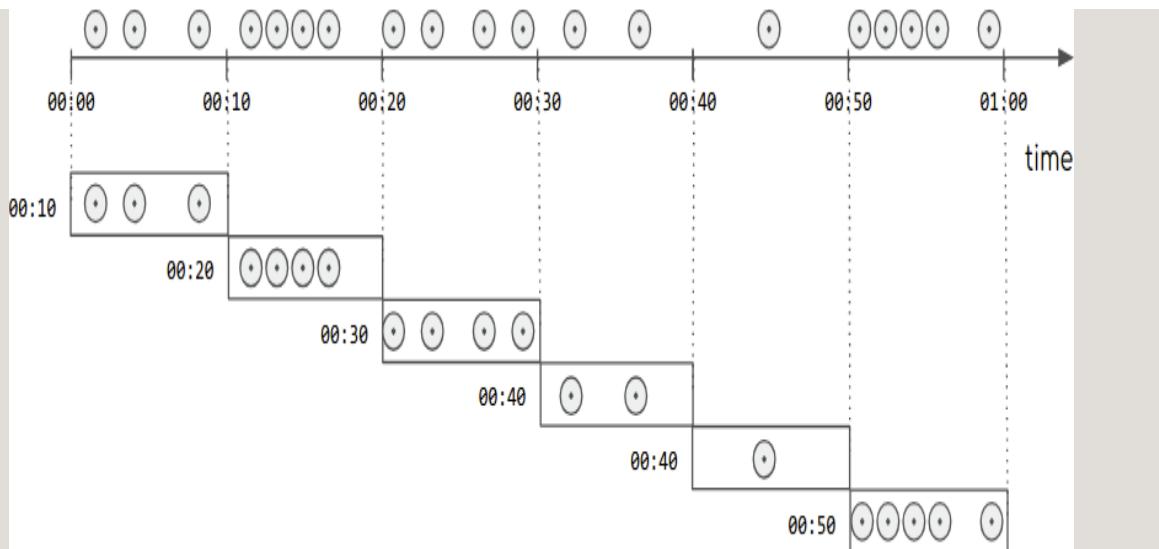


Figure 2-2. Tumbling windows

Sliding Windows

Sliding windows are aggregates over a period of time that are reported at a higher frequency than the aggregation period itself. As such, sliding windows refer to an aggregation with two time specifications: the window length and the reporting frequency. It usually reads like “a grouping function over a time interval x reported each y frequency.” For example, “the average share price over the last day reported hourly.” As you might have noticed already, this combination of a sliding window with the average function is the most widely known form of a sliding window, commonly known as a *moving average*.

Figure 2-3 shows a sliding window with a window size of 30 seconds and a reporting frequency of 10 seconds. In the illustration, we can observe an important characteristic of *sliding windows*: they are not defined for periods of time smaller than the size of the window. We

can see that there are no windows reported for time 00:10 and 00:20.

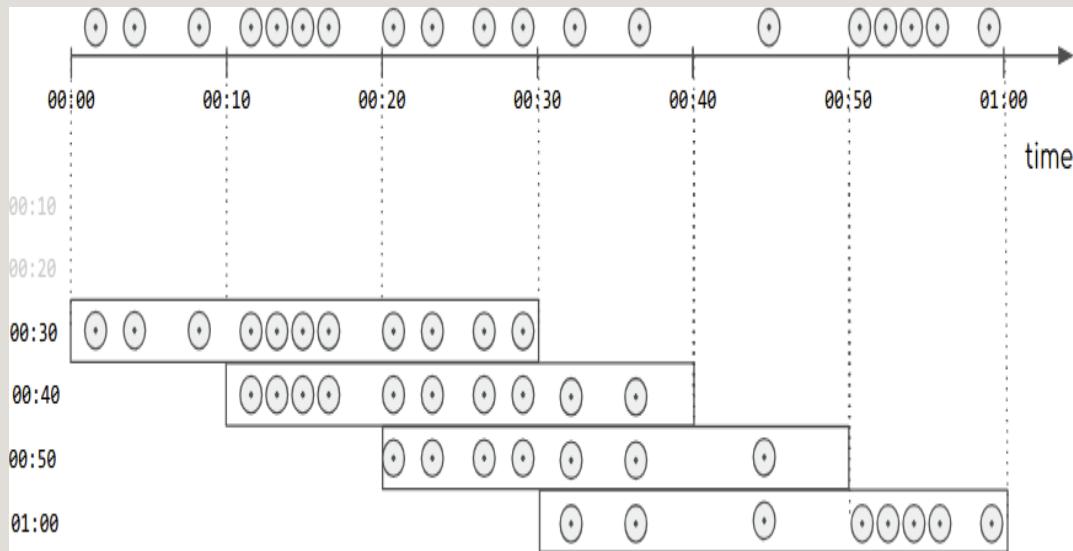


Figure 2-3. Sliding windows

Although you cannot see it in the final illustration, the process of drawing the chart reveals an interesting feature: we can construct and maintain a sliding window by adding the most recent data and removing the expired elements, while keeping all other elements in place.

It's worth noting that tumbling windows are a particular case of sliding windows in which the frequency of reporting is equal to the window size.

Stateless and Stateful Processing

Now that we have a better notion of the programming model of the streaming systems in Apache Spark, we can look at the nature of the computations that we want to apply on data streams. In our context,

data streams are fundamentally long collections of elements, observed over time. In fact, Structured Streaming pushes this logic by considering a data stream as a virtual table of records in which each row corresponds to an element.

Stateful Streams

Whether streams are viewed as a continuously extended collection or as a table, this approach gives us some insight into the kind of computation that we might find interesting. In some cases, the emphasis is put on the continuous and independent processing of elements or groups of elements: those are the cases for which we want to operate on some elements based on a well-known heuristic, such as alert messages coming from a log of events.

This focus is perfectly valid but hardly requires an advanced analytics system such as Apache Spark. More often, we are interested in a real-time reaction to new elements based on an analysis that depends on the whole stream, such as detecting outliers in a collection or computing recent aggregate statistics from event data. For example, it might be interesting to find higher than usual vibration patterns in a stream of airplane engine readings, which requires understanding the regular vibration measurements for the kind of engine we are interested in.

This approach, in which we are simultaneously trying to understand new data in the context of data already seen, often leads us to *stateful stream processing*. Stateful stream processing is the discipline by which we compute something out of the new elements of data

observed in our input data stream and refresh internal data that helps us perform this computation.

For example, if we are trying to do anomaly detection, the internal state that we want to update with every new stream element would be a machine learning model, whereas the computation we want to perform is to say whether an input element should be classified as an anomaly or not.

This pattern of computation is supported by a distributed streaming system such as Apache Spark because it can take advantage of a large amount of computing power and is an exciting new way of reacting to real-time data. For example, we could compute the running mean and standard deviation of the elements seen as input numbers and output a message if a new element is further away than five standard deviations from this mean. This is a simple, but useful, way of marking particular extreme outliers of the distribution of our input elements.¹ In this case, the internal state of the stream processor only stores the running mean and standard deviation of our stream—that is, a couple of numbers.

BOUNDING THE SIZE OF THE STATE

One of the common pitfalls of practitioners new to the field of stream processing is the temptation to store an amount of internal data that is proportional to the size of the input data stream. For example, if you would like to remove duplicate records of a stream, a naive way of approaching that problem would be to store every message already seen and compare new messages to them. That not only increases computing time with each incoming record, but also has an unbounded memory requirement that will eventually outgrow any cluster.

This is a common mistake because the premise of stream processing is that there is no limit to the number of input events and, while your available memory in a distributed Spark cluster might be large, it is always limited. As such, intermediary state representations can be very useful to express computation that operates on elements relative to the global stream of data on which they are observed, but it is a somewhat unsafe approach. If you choose to have intermediate data, you need to make absolutely sure that the amount of data you might be storing at any given time is strictly bounded to a certain upper limit that is less than your available memory, independent of the amount of data that you might encounter as input.

An Example: Local Stateful Computation in Scala

To gain intuition into the concept of statefulness without having to go into the complexity of distributed stream processing, we begin with a simple nondistributed stream example in Scala.

The Fibonacci Sequence is classically defined as a stateful stream: it's the sequence starting with 0 and 1, and thereafter composed of the sum of its two previous elements, as shown in Example 2-1.

Example 2-1. A stateful computation of the Fibonacci elements

```
scala> val ints = Stream.from(0)
ints: scala.collection.immutable.Stream[Int] = Stream(0, ?)

scala> val fibs = (ints.scanLeft((0, 1)){ case ((previous,
current), index) =>
    (current, (previous + current))})

fibs: scala.collection.immutable.Stream[(Int, Int)] =
Stream((0,1), ?)

scala> fibs.take(8).print
(0,1), (1,1), (1,2), (2,3), (3,5), (5,8), (8,13), (13,21),
empty

Scala> fibs.map{ case (x, y) => x}.take(8).print
0, 1, 1, 2, 3, 5, 8, 13, empty
```

Stateful stream processing refers to any stream processing that looks to past information to obtain its result. It's necessary to maintain some *state* information in the process of computing the next element of the stream.

Here, this is held in the recursive argument of the `scanLeft` function, in which we can see `fibs` having a tuple of two elements for each element: the sought result, and the next value. We can apply a simple transformation to the list of tuples `fibs` to retain only the leftmost element and thus obtain the classical Fibonacci Sequence.

The important point to highlight is that to get the value at the *nth* place, we must process all *n-1* elements, keeping the intermediate (*i-1*, *i*) elements as we move along the stream.

Would it be possible to define it without referring to its prior values, though, purely statelessly?

A Stateless Definition of the Fibonacci Sequence as a Stream Transformation

To express this computation as a stream, taking as input the integers and outputting the Fibonacci Sequence, we express this as a stream transformation that uses a stateless map function to transform each number to its Fibonacci value. We can see the implementation of this approach in Example 2-2.

Example 2-2. A stateless computation of the Fibonacci elements

```
scala> import scala.math.{pow, sqrt}
import scala.math.{pow, sqrt}

scala> val phi = (sqrt(5)+1) / 2
phi: Double = 1.618033988749895

scala> def fibonacciNumber(x: Int): Int =
    ((pow(phi, x) - pow(-phi, -x))/sqrt(5)).toInt
fibonacciNumber: (x: Int)Int

scala> val integers = Stream.from(0)
integers: scala.collection.immutable.Stream[Int] = Stream(0,
?)
scala> integers.take(10).print
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, empty

scala> val fibonacciSequence = integers.map(fibonacciNumber)
fibonacciSequence: scala.collection.immutable.Stream[Int] =
Stream(0, ?)

scala> fibonacciSequence.take(8).print
0, 1, 1, 2, 3, 5, 8, 13, empty
```

This rather counterintuitive definition uses a stream of integers, starting from the single integer (0), to then define the Fibonacci Sequence as a computation that takes as input an integer n received over the stream and returns the n -th element of the Fibonacci Sequence as a result. This uses a floating-point number formula

known as the *Binet formula* to compute the n -th element of the sequence directly, without requiring the previous elements; that is, without requiring knowledge of the state of the stream.

Notice how we take a limited number of elements of this sequence and print them in Scala, as an explicit operation. This is because the computation of elements in our stream is executed lazily, which calls the evaluation of our stream only when required, considering the elements needed to produce them from the last materialization point to the original source.

Stateless or Stateful Streaming

We illustrated the difference between stateful and stateless stream processing with a rather simple case that has a solution using the two approaches. Although the stateful version closely resembles the definition, it requires more computing resources to produce a result: it needs to traverse a stream and keep intermediate values at each step.

The stateless version, although contrived, uses a simpler approach: we use a stateless function to obtain a result. It doesn't matter whether we need the Fibonacci number for 9 or 999999, in both cases the computational cost is roughly of the same order.

We can generalize this idea to stream processing. Stateful processing is more costly in terms of the resources it uses and also introduces concerns in face of failure: what happens if our computation fails halfway through the stream? Although a safe rule of thumb is to choose for the stateless option, if available, many of the interesting

questions we can ask over a stream of data are often stateful in nature. For example: how long was the user session on our site? What was the path the taxi used across the city? What is the moving average of the pressure sensor on an industrial machine?

Throughout the book, we will see that stateful computations are more general but they carry their own constraints. It's an important aspect of the stream-processing framework to provide facilities to deal with these constraints and free the user to create the solutions that the business needs dictate.

The Effect of Time

So far, we have considered how there is an advantage in keeping track of intermediary data as we produce results on each element of the data stream because it allows us to analyze each of those elements relative to the data stream that they are part of as long as we keep this intermediary data of a bounded and reasonable size. Now, we want to consider another issue unique to stream processing, which is the operation on time-stamped messages.

Computing on Timestamped Events

Elements in a data stream always have a *processing time*. That is, by definition, the time at which the stream-processing system observes a new event from a data source. That time is entirely determined by the processing runtime and completely independent of the content of the stream's element.

However, for most data streams, we also speak of a notion of *event time*, which is the time when the event actually happened. When the capabilities of the system sensing the event allow for it, this time is usually added as part of the message payload in the stream.

Timestamping is an operation that consists of adding a register of time at the moment of the generation of the message, which will become a part of the data stream. It is a ubiquitous practice that is present in both the most humble embedded devices (provided they have a clock) as well as the most complex logs in financial transaction systems.

Timestamps as the Provider of the Notion of Time

The importance of time stamping is that it allows users to reason on their data considering the moment at which it was generated.

For example, if I register my morning jog using a wearable device and I synchronize the device to my phone when I get back home, I would like to see the details of my heart rate and speed as I ran through the forest moments ago, and not see the data as a timeless sequence of values as they are being uploaded to some cloud server. As we can see, timestamps provide the context of time to data.

So, because event logs form a large proportion of the data streams being analyzed today, those timestamps help make sense of what happened to a particular system at a given time. This complete picture is something that is often made more elusive by the fact that transporting the data from the various systems or devices that have

created it to the cluster that processes it is an operation prone to different forms of failure in which some events could be delayed, reordered, or lost.

Often, users of a framework such as Apache Spark want to compensate for those hazards without having to compromise on the reactivity of their system. Out of this desire was born a discipline for producing the following:

- Clearly marked correct and reordered results
- Intermediary prospective results

With that classification reflecting the best knowledge that a stream-processing system has of the timestamped events delivered by the data stream and under the proviso that this view could be completed by the late arrival of delayed stream elements. This process constitutes the basis for *event-time processing*.

In Spark, this feature is offered natively only by Structured Streaming. Even though Spark Streaming lacks built-in support for event-time processing, it is a question of development effort and some data consolidation processes to manually implement the same sort of primitives, as you will see in Chapter 22.

Event Time Versus Processing Time

We recognize that there is a timeline in which the events are created and a different one when they are processed:

- *Event time* refers to the timeline when the events were originally generated. Typically, a clock available at the generating device places a timestamp in the event itself, meaning that all events from the same source could be chronologically ordered even in the case of transmission delays.
- *Processing time* is the time when the event is handled by the stream-processing system. This time is relevant only at the technical or implementation level. For example, it could be used to add a processing timestamp to the results and in that way, differentiate duplicates, as being the same output values with different processing times.

Imagine that we have a series of events produced and processed over time, as illustrated in Figure 2-4.

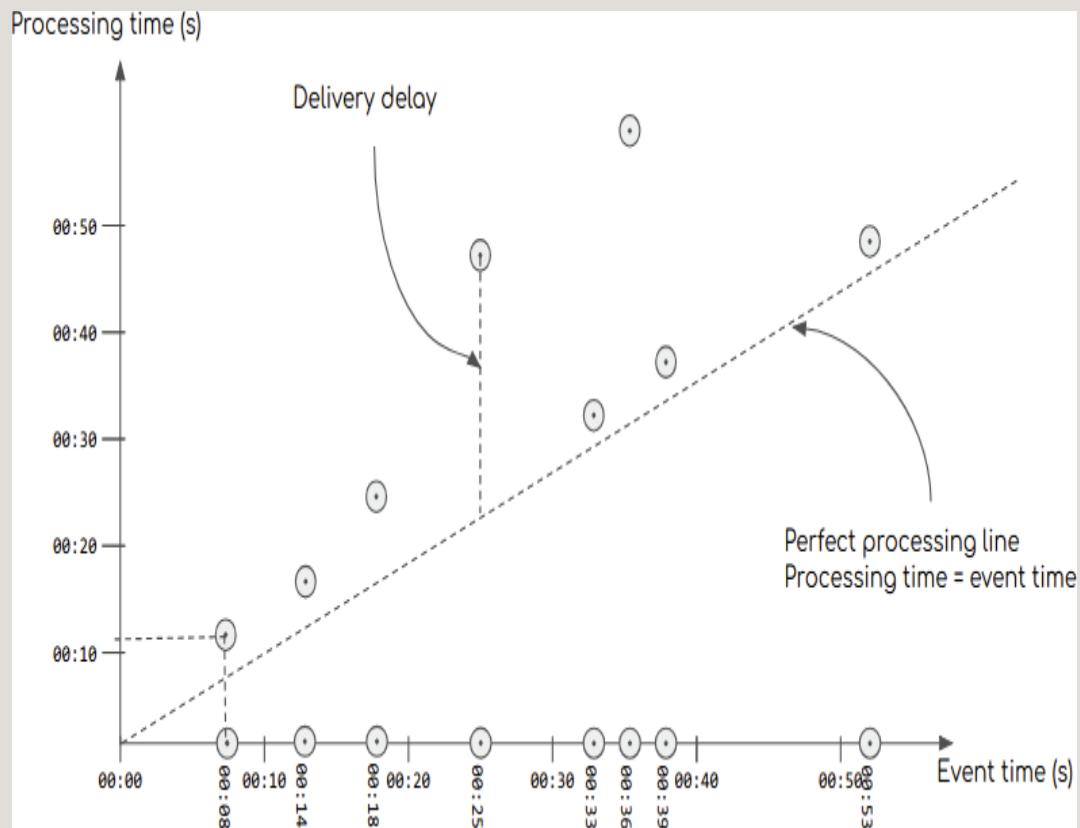


Figure 2-4. Event versus processing time

Let's look at this more closely:

- The x-axis represents the event timeline and the dots on that axis denote the time at which each event was generated.
- The y-axis is the processing time. Each dot on the chart area corresponds to when the corresponding event in the x-axis is processed. For example, the event created at 00:08 (first on the x-axis) is processed at approximately 00:12, the time that corresponds to its mark on the y-axis.
- The diagonal line represents the perfect processing time. In an ideal world, using a network with zero delay, events are processed immediately as they are created. Note that there can be no processing events below that line, because it would mean that events are processed before they are created.
- The vertical distance between the diagonal and the processing time is the *delivery delay*: the time elapsed between the production of the event and its eventual consumption.

With this framework in mind, let's now consider a 10-second window aggregation, as demonstrated in Figure 2-5.

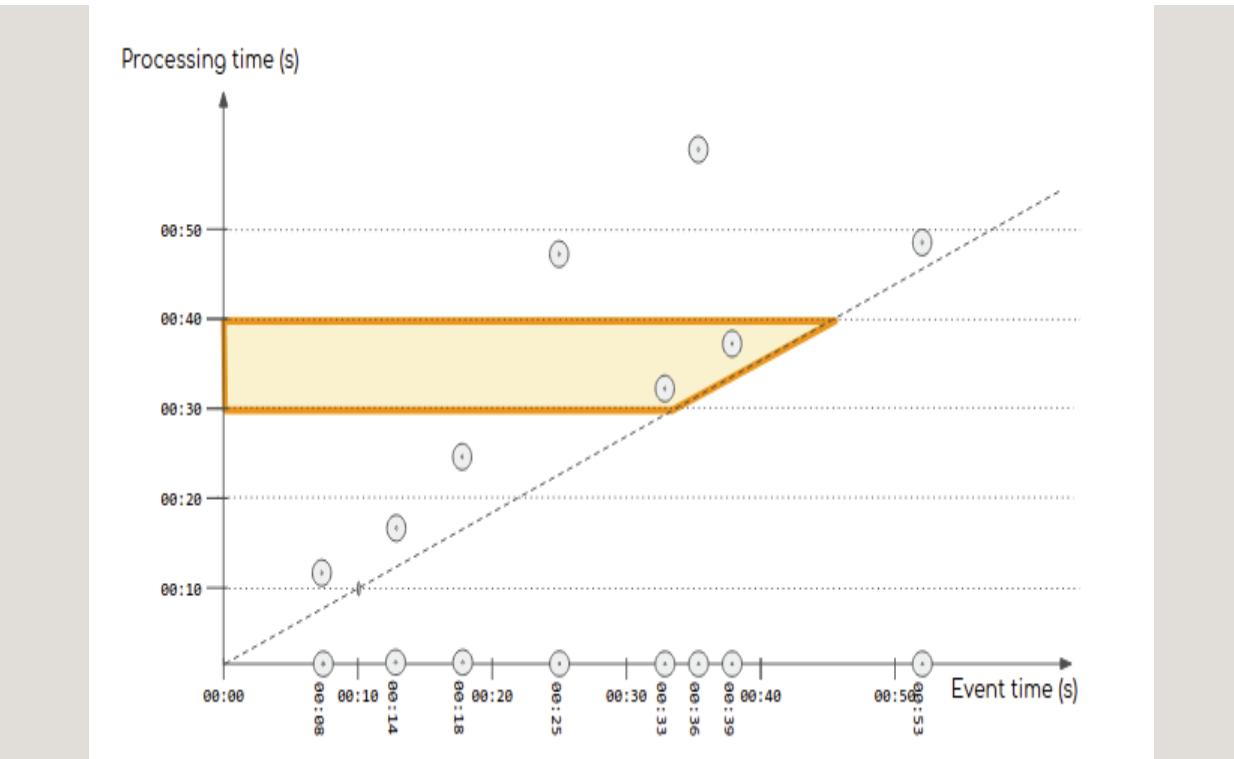


Figure 2-5. Processing-time windows

We start by considering windows defined on processing time:

- The stream processor uses its internal clock to measure 10-second intervals.
- All events that fall in that time interval belong to the window.
- In Figure 2-5, the horizontal lines define the 10-second windows.

We have also highlighted the window corresponding to the time interval 00:30–00:40. It contains two events with event time 00:33 and 00:39.

In this window, we can appreciate two important characteristics:

- The window boundaries are well defined, as we can see in the highlighted area. This means that the window has a defined start and end. We know what's in and what's out by the time the window closes.
- Its contents are arbitrary. They are unrelated to when the events were generated. For example, although we would assume that a $00:30 - 00:40$ window would contain the event $00:36$, we can see that it has fallen out of the resulting set because it was late.

Let's now consider the same 10-second window defined on event time. For this case, we use the *event creation time* as the window aggregation criteria. Figure 2-6 illustrates how these windows look radically different from the processing-time windows we saw earlier. In this case, the window $00:30 - 00:40$ contains all the events that were *created* in that period of time. We can also see that this window has no natural upper boundary that defines when the window ends. The event created at $00:36$ was late for more than 20 seconds. As a consequence, to report the results of the window $00:30 - 00:40$, we need to wait at least until $01:00$. What if an event is dropped by the network and never arrives? How long do we wait? To resolve this problem, we introduce an arbitrary deadline called a *watermark* to deal with the consequences of this open boundary, like lateness, ordering, and deduplication.

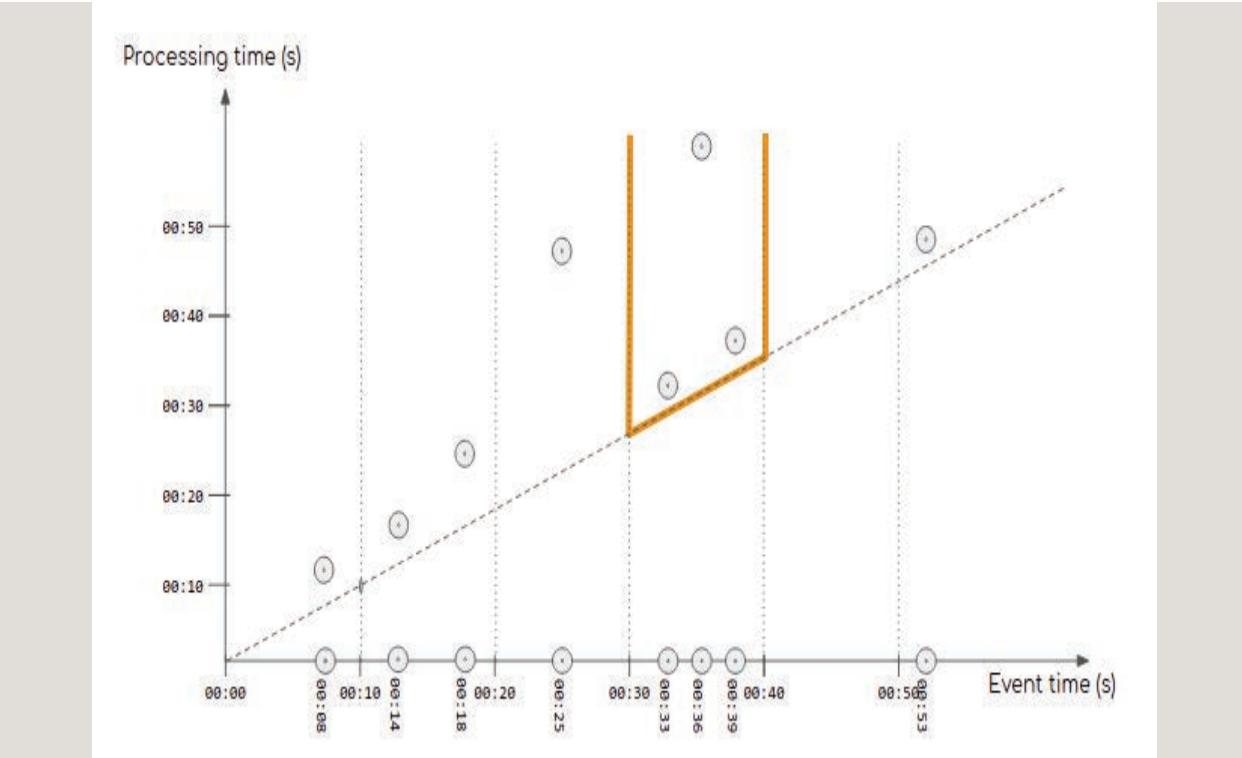


Figure 2-6. Event-time windows

Computing with a Watermark

As we have noticed, stream processing produces periodic results out of the analysis of the events observed in its input. When equipped with the ability to use the timestamp contained in the event messages, the stream processor is able to bucket those messages into two categories based on a notion of a watermark.

The watermark is, at any given moment, *the oldest timestamp that we will accept on the data stream*. Any events that are older than this expectation are not taken into the results of the stream processing. The streaming engine can choose to process them in an alternative way, like report them in a *late arrivals* channel, for example.

However, to account for possibly delayed events, this watermark is usually much larger than the average delay we expect in the delivery of the events. Note also that this watermark is a fluid value that monotonically increases over time,² sliding a window of delay-tolerance as the time observed in the data-stream progresses.

When we apply this concept of watermark to our event-time diagram, as illustrated in Figure 2-7, we can appreciate that the watermark closes the open boundary left by the definition of event-time window, providing criteria to decide what events belong to the window, and what events are too late to be considered for processing.

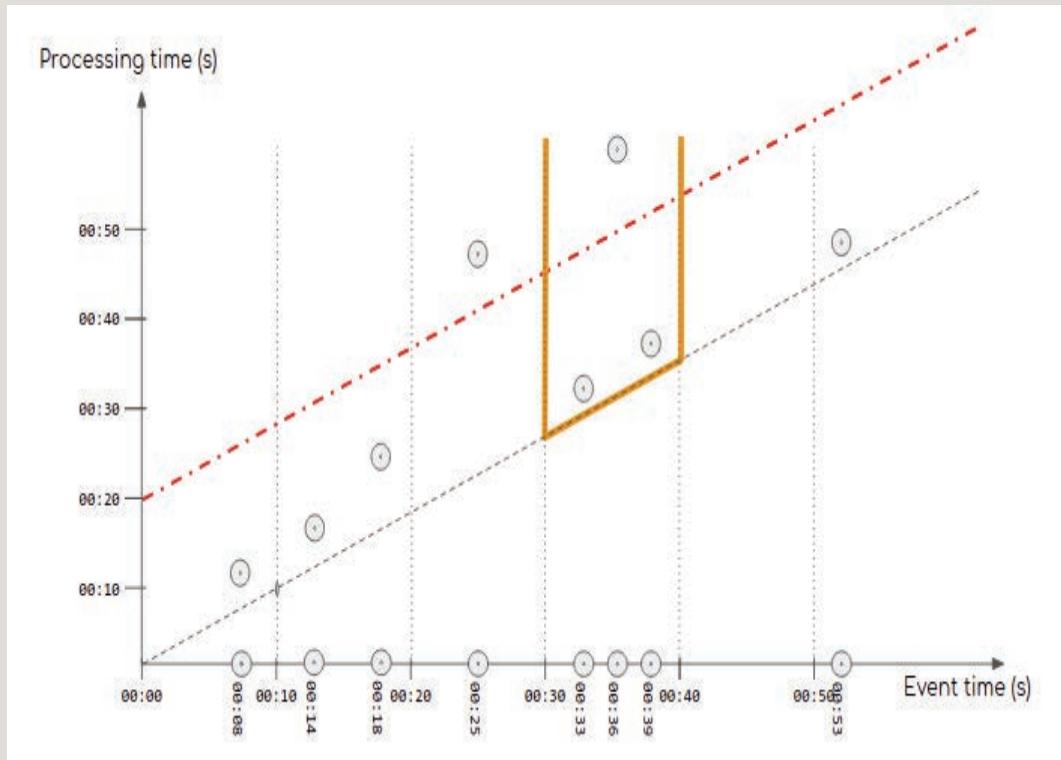


Figure 2-7. Watermark in event time

After this notion of watermark is defined for a stream, the stream processor can operate in one of two modes with relation to that

specific stream: either it is producing output relative to events that are all older than the watermark, in which case the output is final because all of those elements have been observed so far, and no further event older than that will ever be considered, or it is producing an output relative to the data that is before the watermark and a new, delayed element newer than the watermark could arrive on the stream at any moment and can change the outcome. In this latter case, we can consider the output as provisional because newer data can still change the final outcome, whereas in the former case, the result is final and no new data will be able to change it.

We examine in detail how to concretely express and operate this sort of computation in Chapter 12.

Note finally that with provisional results, we are storing intermediate values and in one way or another, we require a method to revise their computation upon the arrival of delayed events. This process requires some amount of memory space. As such, event-time processing is another form of stateful computation and is subject to the same limitation: to handle watermarks, the stream processor needs to store a lot of intermediate data and, as such, consume a significant amount of memory that roughly corresponds to *the length of the watermark × the rate of arrival × message size*.

Also, since we need to wait for the watermark to expire to be sure that we have all elements that comprise an interval, stream processes that use a watermark and want to have a unique, final result for each interval, must delay their output for at least the length of the watermark.

CAUTION

We want to outline event-time processing because it is an exception to the general rule we have given in Chapter 1 of making no assumptions on the throughput of events observed in its input stream.

With event-time processing, we make the assumption that setting our watermark to a certain value is appropriate. That is, we can expect the results of a streaming computation based on event-time processing to be meaningful only if the watermark allows for the delays that messages of our stream will actually encounter between their creation time and their order of arrival on the input data stream.

A too small watermark will lead to dropping too many events and produce severely incomplete results. A too large watermark will delay the output of results deemed complete for too long and increase the resource needs of the stream processing system to preserve all intermediate events.

It is left to the users to ensure they choose a watermark suitable for the event-time processing they require and appropriate for the computing resources they have available, as well.

Summary

In this chapter, we explored the main notions unique to the stream-processing programming model:

- Data sources and sinks
- Stateful processing
- Event-time processing

We explore the implementation of these concepts in the streaming APIs of Apache Spark as we progress through the book.

- 1 Thanks to the Chebycheff inequality, we know that alerts on this data stream should occur with less than 5% probability.
- 2 Watermarks are nondecreasing by nature.

Chapter 3. Streaming Architectures

The implementation of a distributed data analytics system has to deal with the management of a pool of computational resources, as in-house clusters of machines or reserved cloud-based capacity, to satisfy the computational needs of a division or even an entire company. Since teams and projects rarely have the same needs over time, clusters of computers are best amortized if they are a shared resource among a few teams, which requires dealing with the problem of multitenancy.

When the needs of two teams differ, it becomes important to give each a fair and secure access to the resources for the cluster, while making sure the computing resources are best utilized over time.

This need has forced people using large clusters to address this heterogeneity with modularity, making several functional blocks emerge as interchangeable pieces of a data platform. For example, when we refer to database storage as the functional block, the most common component that delivers that functionality is a relational database such as PostgreSQL or MySQL, but when the streaming application needs to write data at a very high throughput, a scalable column-oriented database like Apache Cassandra would be a much better choice.

In this chapter, we briefly explore the different parts that comprise the architecture of a streaming data platform and see the position of a processing engine relative to the other components needed for a complete solution. After we have a good view of the different elements in a streaming architecture, we explore two architectural styles used to approach streaming applications: the Lambda and the Kappa architectures.

Components of a Data Platform

We can see a data platform as a composition of standard components that are expected to be useful to most stakeholders and specialized systems that serve a purpose specific to the challenges that the business wants to address.

Figure 3-1 illustrates the pieces of this puzzle.

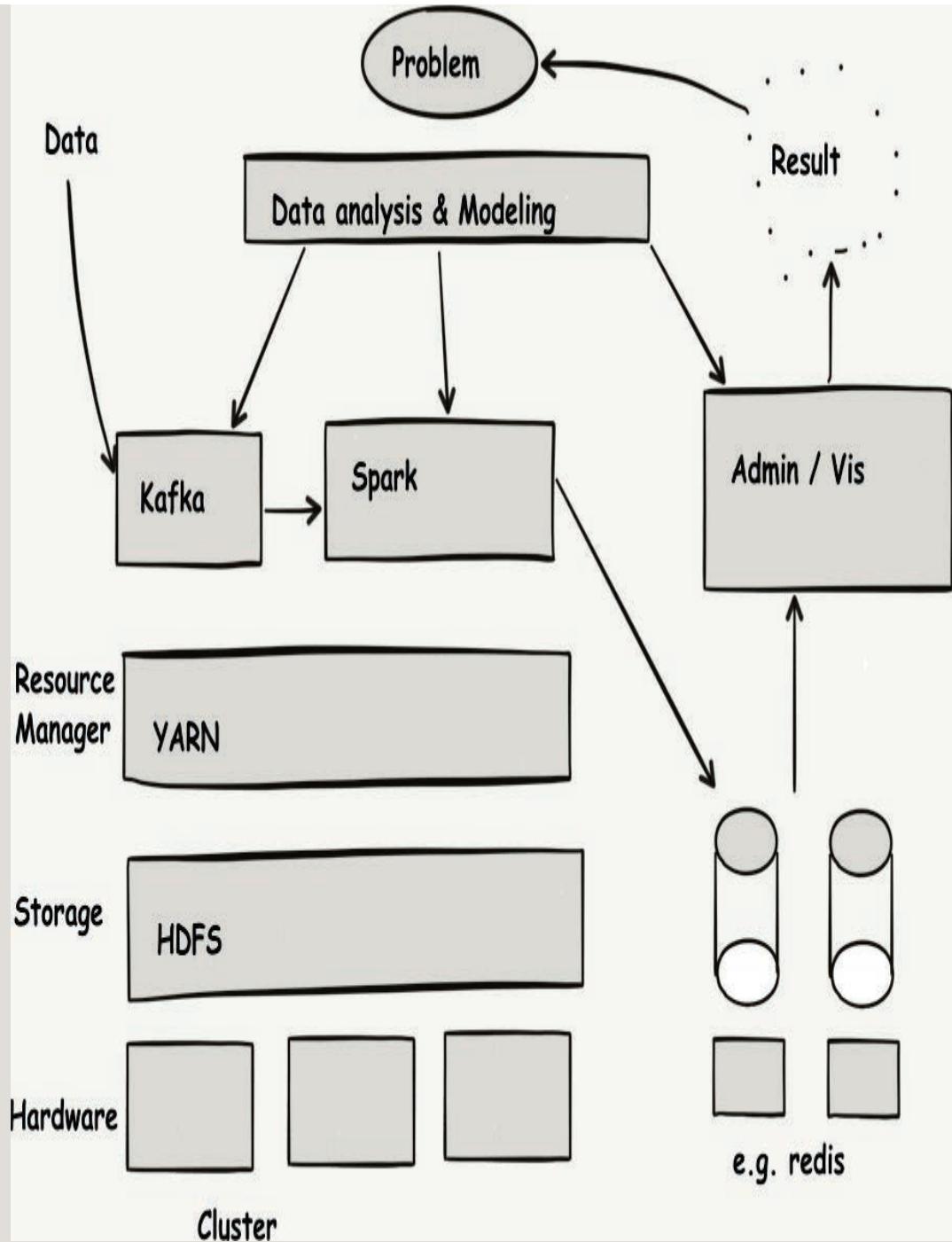


Figure 3-1. The parts of a data platform

Going from the bare-metal level at the bottom of the schema to the actual data processing demanded by a business requirement, you could find the following:

The hardware level

On-premises hardware installations, datacenters, or potentially virtualized in homogeneous cloud solutions (such as the T-shirt size offerings of Amazon, Google, or Microsoft), with a base operating system installed.

The persistence level

On top of that baseline infrastructure, it is often expected that machines offer a shared interface to a persistence solution to store the results of their computation as well as perhaps its input. At this level, you would find distributed storage solutions like the Hadoop Distributed File System (HDFS)—among many other distributed storage systems. On the cloud, this persistence layer is provided by a dedicated service such as Amazon Simple Storage Service (Amazon S3) or Google Cloud Storage.

The resource manager

After persistence, most cluster architectures offer a single point of negotiation to submit jobs to be executed on the cluster. This is the task of the resource manager, like YARN and Mesos, and the more evolved *schedulers* of the *cloud-native* era, like Kubernetes.

The execution engine

At an even higher level, there is the execution engine, which is tasked with executing the actual computation. Its defining characteristic is that it holds the interface with the programmer's input and describes the data manipulation. Apache Spark, Apache Flink, or MapReduce would be examples of this.

A data ingestion component

Besides the execution engine, you could find a data ingestion server that could be plugged directly into that engine. Indeed, the old practice of reading data from a distributed filesystem is often

supplemented or even replaced by another data source that can be queried in real time. The realm of messaging systems or log processing engines such as Apache Kafka is set at this level.

A processed data sink

On the output side of an execution engine, you will frequently find a high-level data sink, which might be either another analytics system (in the case of an execution engine tasked with an *Extract, Transform and Load* [ETL] job), a NoSQL database, or some other service.

A visualization layer

We should note that because the results of data-processing are useful only if they are integrated in a larger framework, those results are often plugged into a visualization. Nowadays, since the data being analyzed evolves quickly, that visualization has moved away from the old static report toward more real-time visual interfaces, often using some web-based technology.

In this architecture, Spark, as a computing engine, focuses on providing data processing capabilities and relies on having functional interfaces with the other blocks of the picture. In particular, it implements a cluster abstraction layer that lets it interface with YARN, Mesos, and Kubernetes as resource managers, provides connectors to many data sources while new ones are easily added through an easy-to-extend API, and integrates with output data sinks to present results to upstream systems.

Architectural Models

Now we turn our attention to the link between stream processing and batch processing in a concrete architecture. In particular, we're going

to ask ourselves the question of whether batch processing is still relevant if we have a system that can do stream processing, and if so, why?

In this chapter, we contrast two conceptions of streaming application architecture: the *Lambda architecture*, which suggests duplicating a streaming application with a batch counterpart running in parallel to obtain complementary results, and the *Kappa architecture*, which purports that if two versions of an application need to be compared, those should both be streaming applications. We are going to see in detail what those architectures intend to achieve, and we examine that although the Kappa architecture is easier and lighter to implement in general, there might be cases for which a Lambda architecture is still needed, and why.

The Use of a Batch-Processing Component in a Streaming Application

Often, if we develop a batch application that runs on a periodic interval into a streaming application, we are provided with batch datasets already—and a batch program representing this periodic analysis, as well. In this evolution use case, as described in the prior chapters, we want to evolve to a streaming application to reap the benefits of a lighter, simpler application that gives faster results.

In a greenfield application, we might also be interested in creating a reference batch dataset: most data engineers don't work on merely solving a problem once, but revisit their solution, and continuously improve it, especially if value or revenue is tied to the performance of

their solution. For this purpose, a batch dataset has the advantage of setting a benchmark: after it's collected, it does not change anymore and can be used as a "test set." We can indeed replay a batch dataset to a streaming system to compare its performance to prior iterations or to a known benchmark.

In this context, we identify three levels of interaction between the batch and the stream-processing components, from the least to the most mixed with batch processing:

Code reuse

Often born out of a reference batch implementation, seeks to reemploy as much of it as possible, so as not to duplicate efforts. This is an area in which Spark shines, since it is particularly easy to call functions that transform Resilient Distributed Databases (RDDs) and DataFrames—they share most of the same APIs, and only the setup of the data input and output is distinct.

Data reuse

Wherein a streaming application feeds itself from a feature or data source prepared, at regular intervals, from a batch processing job. This is a frequent pattern: for example, some international applications must handle time conversions, and a frequent pitfall is that daylight saving rules change on a more frequent basis than expected. In this case, it is good to be thinking of this data as a new dependent source that our streaming application feeds itself off.

Mixed processing

Wherein the application itself is understood to have both a batch and a streaming component during its lifetime. This pattern does happen relatively frequently, out of a will to manage both the

precision of insights provided by an application, and as a way to deal with the versioning and the evolution of the application itself.

The first two uses are uses of convenience, but the last one introduces a new notion: using a batch dataset as a benchmark. In the next subsections, we see how this affects the architecture of a streaming application.

Referential Streaming Architectures

In the world of replay-ability and performance analysis over time, there are two historical but conflicting recommendations. Our main concern is about how to measure and test the performance of a streaming application. When we do so, there are two things that can change in our setup: the nature of our model (as a result of our attempt at improving it) and the data that the model operates on (as a result of organic change). For instance, if we are processing data from weather sensors, we can expect a seasonal pattern of change in the data.

To ensure we compare apples to apples, we have already established that replaying a *batch dataset* to two versions of our streaming application is useful: it lets us make sure that we are not seeing a change in performance that is really reflecting a change in the data. Ideally, in this case, we would test our improvements in yearly data, making sure we're not overoptimizing for the current season at the detriment of performance six months after.

However, we want to contend that a comparison with a *batch analysis* is necessary, as well, beyond the use of a benchmark dataset—and this is where the architecture comparison helps.

The Lambda Architecture

The Lambda architecture (Figure 3-2) suggests taking a batch analysis performed on a periodic basis—say, nightly—and to supplement the model thus created with streaming refinements as data comes, until we are able to produce a new version of the batch analysis based on the entire day's data.

It was introduced as such by Nathan Marz in a blog post, “[How to beat the CAP Theorem](#)”.¹ It proceeds from the idea that we want to emphasize two novel points beyond the precision of the data analysis:

- The historical replay-ability of data analysis is important
- The availability of results proceeding from fresh data is also a very important point

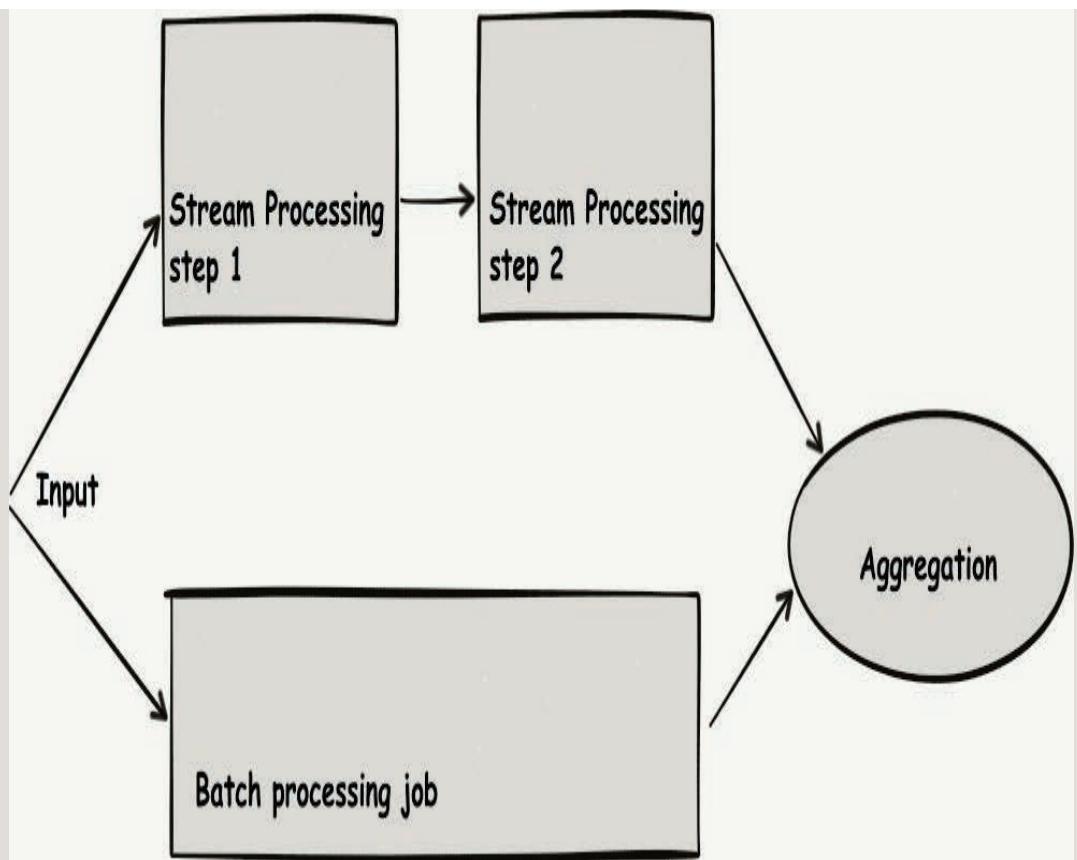


Figure 3-2. The Lambda architecture

This is a useful architecture, but its drawbacks seem obvious, as well: such a setup is complex and requires maintaining two versions of the same code, for the same purpose. Even if Spark helps in letting us reuse most of our code between the batch and streaming versions of our application, the two versions of the application are distinct in life cycles, which might seem complicated.

An alternative view on this problem suggests that it would be enough to keep the ability to feed the same dataset to two versions of a streaming application (the new, improved experiment, and the older, stable workhorse), helping with the maintainability of our solution.

The Kappa Architecture

This architecture, as outlined in Figure 3-3, compares two streaming applications and does away with any batching, noting that if reading a batch file is needed, a simple component can replay the contents of this file, record by record, as a streaming data source. This simplicity is still a great benefit, since even the code that consists in feeding data to the two versions of this application can be reused. In this paradigm, called the *Kappa architecture* ([Kreps2014]), there is no deduplication and the mental model is simpler.

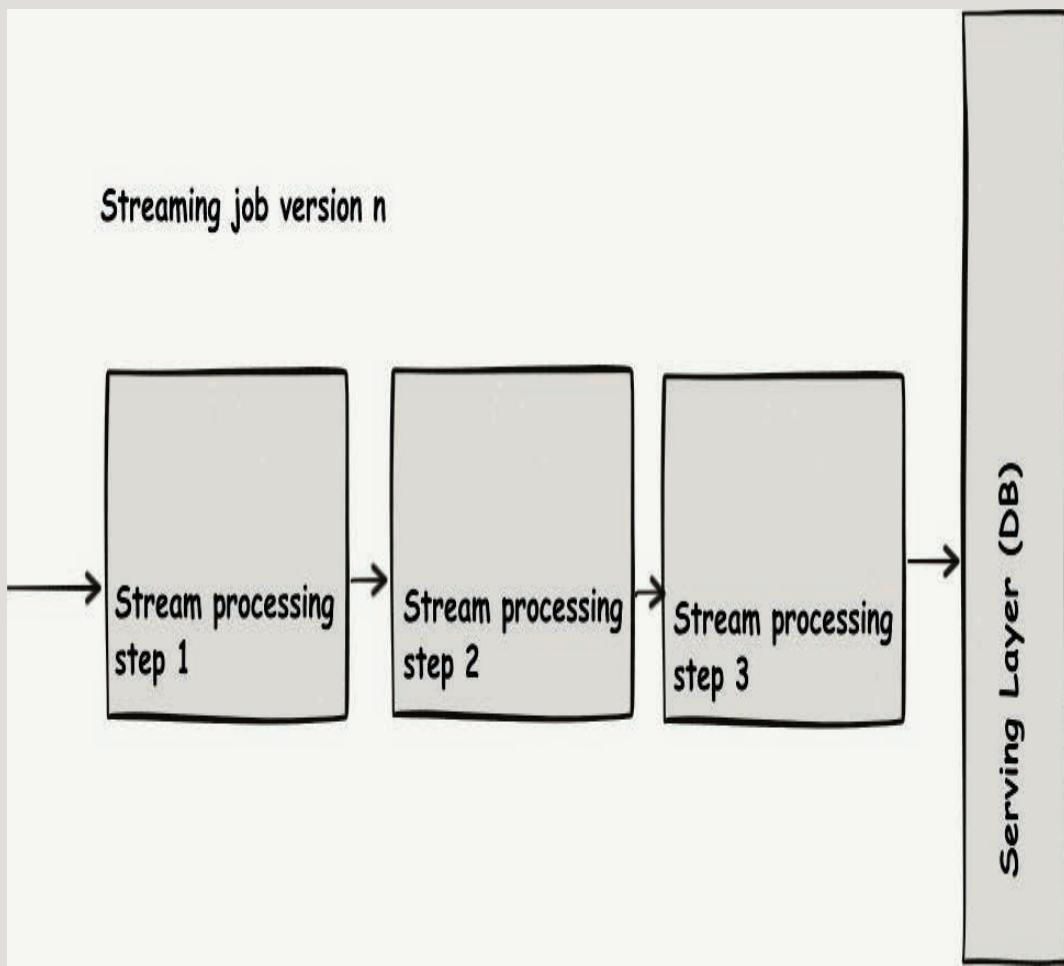


Figure 3-3. The Kappa architecture

This begs the question: is batch computation still relevant? Should we convert our applications to be all streaming, all the time?

We think some concepts stemming from the Lambda architecture are still relevant; in fact, they're vitally useful in some cases, although those are not always easy to figure out.

There are some use cases for which it is still useful to go through the effort of implementing a batch version of our analysis and then compare it to our streaming solution.

Streaming Versus Batch Algorithms

There are two important considerations that we need to take into account when selecting a general architectural model for our streaming application:

- Streaming algorithms are sometimes completely different in nature
- Streaming algorithms can't be guaranteed to measure well against batch algorithms

Let's explore these thoughts in the next two sections using motivating examples.

Streaming Algorithms Are Sometimes Completely Different in Nature

Sometimes, it is difficult to deduce batch from streaming, or the reverse, and those two classes of algorithms have different characteristics. This means that at first glance we might not be able to reuse code between both approaches, but also, and more important,

that relating the performance characteristics of those two modes of processing should be done with high care.

To make things more precise, let's look at an example: the buy or rent problem. In this case, we decide to go skiing. We can buy skis for \$500 or rent them for \$50. Should we rent or buy?

Our intuitive strategy is to first rent, to see if we like skiing. But suppose we do: in this case, we will eventually realize we will have spent more money than we would have if we had bought the skis in the first place.

In the batch version of this computation, we proceed “in hindsight,” being given the total number of times we will go skiing in a lifetime. In the streaming, or online version of this problem, we are asked to make a decision (produce an output) on each discrete skiing event, as it happens. The strategy is fundamentally different.

In this case, we can consider the competitive ratio of a streaming algorithm. We run the algorithm on the worst possible input, and then compare its “cost” to the decision that a batch algorithm would have taken, “in hindsight.”

In our buy-or-rent problem, let's consider the following streaming strategy: we rent until renting makes our total spending as much as buying, in which case we buy.

If we go skiing nine times or fewer, we are optimal, because we spend as much as what we would have in hindsight. The competitive

ratio is one. If we go skiing 10 times or more, we pay $\$450 + \$500 = \$950$. The worst input is to receive 10 “ski trip” decision events, in which case the batch algorithm, in hindsight, would have paid \$500. The competitive ratio of this strategy is $(2 - 1/10)$.

If we were to choose another algorithm, say “always buy on the first occasion,” then the worst possible input is to go skiing only once, which means that the competitive ratio is $\$500 / \$50 = 10$.

NOTE

The performance ratio or competitive ratio is a measure of how far from the optimal the values returned by an algorithm are, given a measure of optimality. An algorithm is formally ρ -competitive if its objective value is no more than ρ times the optimal offline value for all instances.

A better competitive ratio is smaller, whereas a competitive ratio above one shows that the streaming algorithm performs measurably worse on some inputs. It is easy to see that with the worst input condition, the batch algorithm, which proceeds in hindsight with strictly more information, is always expected to perform better (the competitive ratio of any streaming algorithm is greater than one).

Streaming Algorithms Can’t Be Guaranteed to Measure Well Against Batch Algorithms

Another example of those unruly cases is the bin-packing problem. In the bin-packing problem, an input of a set of objects of different sizes or different weights must be fitted into a number of bins or

containers, each of them having a set volume or set capacity in terms of weight or size. The challenge is to find an assignment of objects into bins that minimizes the number of containers used.

In computational complexity theory, the offline ration of that algorithm is known to be *NP-hard*. The simple variant of the problem is the *decision* question: knowing whether that set of objects will fit into a specified number of bins. It is itself NP-complete, meaning (for our purposes here) computationally very difficult in and of itself.

In practice, this algorithm is used very frequently, from the shipment of actual goods in containers, to the way operating systems match memory allocation requests, to blocks of free memory of various sizes.

There are many variations of these problems, but we want to focus on the distinction between online versions—for which the algorithm has as input a stream of objects—and offline versions—for which the algorithm can examine the entire set of input objects before it even starts the computing process.

The online algorithm processes the items in arbitrary order and then places each item in the first bin that can accommodate it, and if no such bin exists, it opens a new bin and puts the item within that new bin. This greedy approximation algorithm always allows placing the input objects into a set number of bins that is, at worst, suboptimal; meaning we might use more bins than necessary.

A better algorithm, which is still relatively intuitive to understand, is the *first fit decreasing strategy*, which operates by first sorting the items to be inserted in decreasing order of their sizes, and then inserting each item into the first bin in the list with sufficient remaining space. That algorithm was proven in 2007 to be much closer to the optimal algorithm producing the absolute minimum number of bins ([Dosa2007]).

The first fit decreasing strategy, however, relies on the idea that we can first sort the items in decreasing order of sizes before we begin processing them and packing them into bins.

Now, attempting to apply such a method in the case of the online bin-packing problem, the situation is completely different in that we are dealing with a stream of elements for which sorting is not possible. Intuitively, it is thus easy to understand that the online bin-packing problem—which by its nature lacks foresight when it operates—is much more difficult than the offline bin-packing problem.

WARNING

That intuition is in fact supported by proof if we consider the competitive ratio of streaming algorithms. This is the ratio of resources consumed by the online algorithm to those used by an online optimal algorithm delivering the minimal number of bins by which the input set of objects encountered so far can be packed. This competitive ratio for the knapsack (or bin-packing) problem is in fact arbitrarily bad (that is, large; see [Sharp2007]), meaning that it is always possible to encounter a “bad” sequence in which the performance of the online algorithm will be arbitrarily far from that of the optimal algorithm.

The larger issue presented in this section is that there is no guarantee that a streaming algorithm will perform better than a batch algorithm, because those algorithms must function without foresight. In particular, some online algorithms, including the knapsack problem, have been proven to have an arbitrarily large performance ratio when compared to their offline algorithms.

What this means, to use an analogy, is that we have one worker that receives the data as batch, as if it were all in a *storage room* from the beginning, and the other worker receiving the data in a streaming fashion, as if it were on a *conveyor belt*, then *no matter how clever our streaming worker is, there is always a way to place items on the conveyor belt in such a pathological way that he will finish his task with an arbitrarily worse result than the batch worker*.

The takeaway message from this discussion is twofold:

- Streaming systems are indeed “lighter”: their semantics can express a lot of low-latency analytics in expressive terms.
- Streaming APIs invite us to implement analytics using streaming or online algorithms in which heuristics are sadly limited, as we’ve seen earlier.

Summary

In conclusion, the news of batch processing’s demise is overrated: batch processing is still relevant, at least to provide a baseline of performance for a streaming problem. Any responsible engineer should have a good idea of the performance of a batch algorithm

operating “in hindsight” on the same input as their streaming application:

- If there is a known competitive ratio for the streaming algorithm at hand, and the resulting performance is acceptable, running just the stream processing might be enough.
- If there is no known competitive ratio between the implemented stream processing and a batch version, running a batch computation on a regular basis is a valuable benchmark to which to hold one’s application.

¹ We invite you to consult the original article if you want to know more about the link with the CAP theorem (also called Brewer’s theorem). The idea was that it concentrated some limitations fundamental to distributed computing described by the theorem to a limited part of the data-processing system. In our case, we’re focusing on the practical implications of that constraint.

Chapter 4. Apache Spark as a Stream-Processing Engine

In [Chapter 3](#), we pictured a general architectural diagram of a streaming data platform and identified where Spark, as a distributed processing engine, fits in a big data system.

This architecture informed us about what to expect in terms of interfaces and links to the rest of the ecosystem, especially as we focus on stream data processing with Apache Spark. Stream processing, whether in its Spark Streaming or Structured Streaming incarnation, is another *execution mode* for Apache Spark.

In this chapter, we take a tour of the main features that make Spark stand out as a stream-processing engine.

The Tale of Two APIs

As we mentioned in [“Introducing Apache Spark”](#), Spark offers two different stream-processing APIs, Spark Streaming and Structured Streaming:

Spark Streaming

This is an API and a set of connectors, in which a Spark program is being served small batches of data collected from a stream in the form of microbatches spaced at fixed time intervals, performs

a given computation, and eventually returns a result at every interval.

Structured Streaming

This is an API and a set of connectors, built on the substrate of a SQL query optimizer, Catalyst. It offers an API based on DataFrames and the notion of continuous queries over an unbounded table that is constantly updated with fresh records from the stream.

The interface that Spark offers on these fronts is particularly rich, to the point where this book devotes large parts explaining those two ways of processing streaming datasets. One important point to realize is that both APIs rely on the core capabilities of Spark and share many of the low-level features in terms of distributed computation, in-memory caching, and cluster interactions.

As a leap forward from its MapReduce predecessor, Spark offers a rich set of operators that allows the programmer to express complex processing, including machine learning or event-time manipulations. We examine more specifically the basic properties that allow Spark to perform this feat in a moment.

We would just like to outline that these interfaces are by design as simple as their batch counterparts—operating on a DStream feels like operating on an RDD, and operating on a streaming Dataframe looks eerily like operating on a batch one.

Apache Spark presents itself as a unified engine, offering developers a consistent environment whenever they want to develop a batch or a

streaming application. In both cases, developers have all the power and speed of a distributed framework at hand.

This versatility empowers development agility. Before deploying a full-fledged stream-processing application, programmers and analysts first try to discover insights in interactive environments with a fast feedback loop. Spark offers a built-in shell, based on the Scala *REPL* (short for Read-Eval-Print-Loop) that can be used as prototyping grounds. There are several notebook implementations available, like Zeppelin, Jupyter, or the Spark Notebook, that take this interactive experience to a user-friendly web interface. This prototyping phase is essential in the early phases of development, and so is its velocity.

If you refer back to the diagram in Figure 3-1, you will notice that what we called *results* in the chart are actionable insights—which often means revenue or cost-savings—are generated every time a loop (starting and ending at the business or scientific problem) is traveled fully. In sum, this loop is a crude representation of the experimental method, going through observation, hypothesis, experiment, measure, interpretation, and conclusion.

Apache Spark, in its streaming modules, has always made the choice to carefully manage the cognitive load of switching to a streaming application. It also has other major design choices that have a bearing on its stream-processing capabilities, starting with its in-memory storage.

Spark's Memory Usage

Spark offers in-memory storage of slices of a dataset, which must be initially loaded from a data source. The data source can be a distributed filesystem or another storage medium. Spark's form of in-memory storage is analogous to the operation of caching data.

Hence, a *value* in Spark's in-memory storage has a *base*, which is its initial data source, and layers of successive operations applied to it.

Failure Recovery

What happens in case of a failure? Because Spark knows exactly which data source was used to ingest the data in the first place, and because it also knows all the operations that were performed on it thus far, it can reconstitute the segment of lost data that was on a crashed executor, from scratch. Obviously, this goes faster if that reconstitution (*recovery*, in Spark's parlance), does not need to be totally *from scratch*. So, Spark offers a replication mechanism, quite in a similar way to distributed filesystems.

However, because memory is such a valuable yet limited commodity, Spark makes (by default) the cache short lived.

Lazy Evaluation

As you will see in greater detail in later chapters, a good part of the operations that can be defined on values in Spark's storage have a lazy execution, and it is the execution of a final, eager output operation that will trigger the actual execution of computation in a Spark cluster. It's worth noting that if a program consists of a series of linear operations, with the previous one feeding into the next, the

intermediate results *disappear* right after said next step has consumed its input.

Cache Hints

On the other hand, what happens if we have several operations to do on a single intermediate result? Should we have to compute it several times? Thankfully, Spark lets users specify that an intermediate value is important and how its contents should be safeguarded for later.

Figure 4-1 presents the data flow of such an operation.

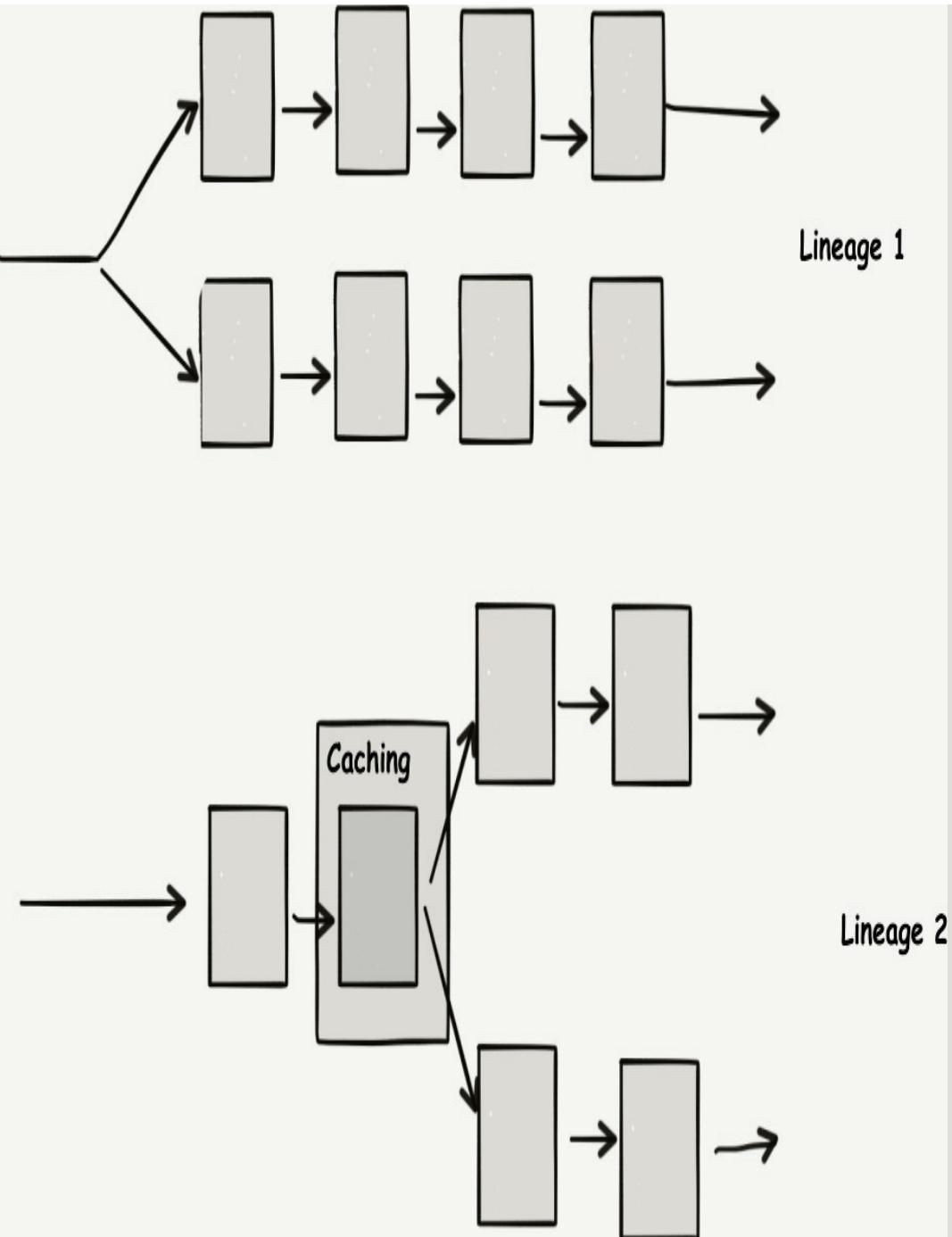


Figure 4-1. Operations on cached values

Finally, Spark offers the opportunity to spill the cache to secondary storage in case it runs out of memory on the cluster, extending the in-memory operation to secondary—and significantly slower—storage

to preserve the functional aspects of a data process when faced with temporary peak loads.

Now that we have an idea of the main characteristics of Apache Spark, let's spend some time focusing on one design choice internal to Spark, namely, the latency versus throughput trade-off.

Understanding Latency

Spark Streaming, as we mentioned, makes the choice of microbatching. It generates a chunk of elements on a fixed interval, and when that interval “tick” elapses, it begins processing the data collected over the last interval. Structured Streaming takes a slightly different approach in that it will make the interval in question as small as possible (the processing time of the last microbatch)—and proposing, in some cases, a continuous processing mode, as well. Yet, nowadays, microbatching is still the dominating internal execution mode of stream processing in Apache Spark.

A consequence of microbatching is that any microbatch delays the processing of any particular element of a batch by at least the time of the batch interval.

Firstly, microbatches create a baseline latency. The jury is still out on how small it is possible to make this latency, though approximately one second is a relatively common number for the lower bound. For many applications, a latency in the space of a few minutes is sufficient; for example:

- Having a dashboard that refreshes you on key performance indicators of your website over the last few minutes
- Extracting the most recent trending topics in a social network
- Computing the energy consumption trends of a group of households
- Introducing new media in a recommendation system

Whereas Spark is an equal-opportunity processor and delays all data elements for (at most) one batch before acting on them, some other streaming engines exist that can fast-track some elements that have priority, ensuring a faster responsivity for them. If your response time is essential for these specific elements, alternative stream processors like Apache Flink or Apache Storm might be a better fit. But if you're just interested in fast processing *on average*, such as when monitoring a system, Spark makes an interesting proposition.

Throughput-Oriented Processing

All in all, where Spark truly excels at stream processing is with throughput-oriented data analytics.

We can compare the microbatch approach to a train: it arrives at the station, waits for passengers for a given period of time and then transports all passengers that boarded to their destination. Although taking a car or a taxi for the same trajectory might allow one passenger to travel faster door to door, the batch of passengers in the train ensures that far more travelers arrive at their destination. The

train offers higher throughput for the same trajectory, at the expense that some passengers must wait until the train departs.

The Spark core engine is optimized for distributed batch processing. Its application in a streaming context ensures that large amounts of data can be processed per unit of time. Spark amortizes the overhead of distributed task scheduling by having many elements to process at once and, as we saw earlier in this chapter, it utilizes in-memory techniques, query optimizations, caching, and even code generation to speed up the transformational process of a dataset.

When using Spark in an end-to-end application, an important constraint is that downstream systems receiving the processed data must also be able to accept the full output provided by the streaming process. Otherwise, we risk creating application bottlenecks that might cause cascading failures when faced with sudden load peaks.

Spark's Polyglot API

We have now outlined the main design foundations of Apache Spark as they affect stream processing, namely a rich API and an in-memory processing model, defined within the model of an execution engine. We have explored the specific streaming modes of Apache Spark, and still at a high level, we have determined that the predominance of microbatching makes us think of Spark as more adapted to throughput-oriented tasks, for which more data yields more quality. We now want to bring our attention to one additional aspect where Spark shines: its programming ecosystem.

Spark was first coded as a Scala-only project. As its interest and adoption widened, so did the need to support different user profiles, with different backgrounds and programming language skills. In the world of scientific data analysis, Python and R are arguably the predominant languages of choice, whereas in the enterprise environment, Java has a dominant position.

Spark, far from being just a library for distributing computation, has become a polyglot framework that the user can interface with using Scala, Java, Python, or the R language. The development language is still Scala, and this is where the main innovations come in.

CAUTION

The coverage of the Java API has for a long time been fairly synchronized with Scala, owing to the excellent Java compatibility offered by the Scala language. And although in Spark 1.3 and earlier versions Python was lagging behind in terms of functionalities, it is now mostly caught up. The newest addition is R, for which feature-completeness is an enthusiastic work in progress.

This versatile interface has let programmers of various levels and backgrounds flock to Spark for implementing their own data analytics needs. The amazing and growing richness of the contributions to the Spark open source project are a testimony to the strength of Spark as a federating framework.

Nevertheless, Spark's approach to best catering to its users' computing goes beyond letting them use their favorite programming language.

Fast Implementation of Data Analysis

Spark's advantages in developing a streaming data analytics pipeline go beyond offering a concise, high-level API in Scala and compatible APIs in Java and Python. It also offers the simple model of Spark as a practical shortcut throughout the development process.

Component reuse with Spark is a valuable asset, as is access to the Java ecosystem of libraries for machine learning and many other fields. As an example, Spark lets users benefit from, for instance, the Stanford CoreNLP library with ease, letting you avoid the painful task of writing a tokenizer. All in all, this lets you quickly prototype your streaming data pipeline solution, getting first results quickly enough to choose the right components at every step of the pipeline development.

Finally, stream processing with Spark lets you benefit from its model of fault tolerance, leaving you with the confidence that faulty machines are not going to bring the streaming application to its knees. If you have enjoyed the automatic restart of failed Spark jobs, you will doubly appreciate that resiliency when running a 24/7 streaming operation.

In conclusion, Spark is a framework that, while making trade-offs in latency, optimizes for building a data analytics pipeline with agility: fast prototyping in a rich environment and stable runtime performance under adverse conditions are problems it recognizes and tackles head-on, offering users significant advantages.

To Learn More About Spark

This book is focused on streaming. As such, we move quickly through the Spark-centric concepts, in particular about batch processing. The most detailed references are [Karau2015] and [Chambers2018].

On a more low-level approach, the official documentation in the [Spark Programming guide](#) is another accessible must-read.

Summary

In this chapter, you learned about Spark and where it came from.

- You saw how Spark extends that model with key performance improvements, notably in-memory computing, as well as how it expands on the API with new higher-order functions.
- We also considered how Spark integrates into the modern ecosystem of big data solutions, including the smaller footprint it focuses on, when compared to its older brother, Hadoop.
- We focused on the streaming APIs and, in particular, on the meaning of their microbatching approach, what uses they are appropriate for, as well as the applications they would not serve well.
- Finally, we considered stream processing in the context of Spark, and how building a pipeline with agility, along with a reliable, fault-tolerant deployment is its best use case.

Chapter 5. Spark's Distributed Processing Model

As a distributed processing system, Spark relies on the availability and addressability of computing resources to execute any arbitrary workload.

Although it's possible to deploy Spark as a standalone distributed system to solve a punctual problem, organizations evolving in their data maturity level are often required to deploy a complete data architecture, as we discussed in [Chapter 3](#).

In this chapter, we want to discuss the interaction of Spark with its computational environment and how, in turn, it needs to adapt to the features and constraints of the environment of choice.

First, we survey the current choices for a cluster manager: YARN, Mesos, and Kubernetes. The scope of a cluster manager goes beyond running data analytics, and therefore, there are plenty of resources available to get in-depth knowledge on any of them. For our purposes, we are going to provide additional details on the cluster manager provider by Spark as a reference.

After you have an understanding of the role of the cluster manager and the way Spark interacts with it, we look into the aspects of fault tolerance in a distributed environment and how the execution model of Spark functions in that context.

With this background, you will be prepared to understand the data reliability guarantees that Spark offers and how they apply to the streaming execution model.

Running Apache Spark with a Cluster Manager

We are first going to look at the discipline of distributing stream processing on a set of machines that collectively form a *cluster*. This set of machines has a general purpose and needs to receive the streaming application's runtime binaries and launching scripts—something known as *provisioning*. Indeed, modern clusters are managed automatically and include a large number of machines in a situation of *multitenancy*, which means that many stakeholders want to access and use the same cluster at various times in the day of a business. The clusters are therefore managed by *cluster managers*.

Cluster managers are pieces of software that receive utilization requests from a number of users, match them to some resources, reserve the resources on behalf of the users for a given duration, and place user applications onto a number of resources for them to use. The challenges of the cluster manager's role include nontrivial tasks such as figuring out the best placements of user requests among a pool of available machines or securely isolating the user applications

if several share the same physical infrastructure. Some considerations where these managers can shine or break include fragmentation of tasks, optimal placement, availability, preemption, and prioritization. Cluster management is, therefore, a discipline in and of itself, beyond the scope of Apache Spark. Instead, Apache Spark takes advantage of existing cluster managers to distribute its workload over a cluster.

Examples of Cluster Managers

Some examples of popular cluster managers include the following:

- Apache YARN, which is a relatively mature cluster manager born out of the Apache Hadoop project
- Apache Mesos, which is a cluster manager based on Linux's container technology, and which was originally the reason for the existence of Apache Spark
- Kubernetes, which is a modern cluster manager born out of service-oriented deployment APIs, originated in practice at Google and developed in its modern form under the flag of the Cloud Native Computing Foundation

Where Spark can sometimes confuse people is that Apache Spark, as a distribution, includes a cluster manager of its own, meaning Apache Spark has the ability to serve as its own particular deployment orchestrator.

In the rest of this chapter we look at the following:

- Spark's own cluster managers and how their *special purpose* means that they take on less responsibility in the domain of

fault tolerance or multitenancy than production cluster managers like Mesos, YARN, or Kubernetes.

- How there is a standard level of *delivery guarantees* expected out of a distributed streaming application, how they differ from one another, and how Spark meets those guarantees.
- How microbatching, a distinctive factor of Spark's approach to stream processing, comes from the decade-old model of *bulk-synchronous processing* (BSP), and paves the evolution path from Spark Streaming to Structured Streaming.

Spark's Own Cluster Manager

Spark has two internal cluster managers:

The *local* cluster manager

This emulates the function of a cluster manager (or resource manager) for testing purposes. It reproduces the presence of a cluster of distributed machines using a threading model that relies on your local machine having only a few available cores. This mode is usually not very confusing because it executes only on the user's laptop.

The *standalone* cluster manager

A relatively simple, Spark-only cluster manager that is rather limited in its availability to slice and dice resource allocation. The standalone cluster manager holds and makes available the entire worker node on which a Spark executor is deployed and started. It also expects the executor to have been predeployed there, and the actual shipping of that *.jar* to a new machine is not within its scope. It has the ability to take over a specific number of executors, which are part of its deployment of worker nodes, and

execute a task on it. This cluster manager is extremely useful for the Spark developers to provide a bare-bones resource management solution that allows you to focus on improving Spark in an environment without any bells and whistles. The standalone cluster manager is not recommended for production deployments.

As a summary, Apache Spark is a *task scheduler* in that what it schedules are *tasks*, units of distribution of computation that have been extracted from the user program. Spark also communicates and is deployed through cluster managers including Apache Mesos, YARN, and Kubernetes, or allowing for some cases its own standalone cluster manager. The purpose of that communication is to reserve a number of *executors*, which are the units to which Spark understands equal-sized amounts of computation resources, a virtual “node” of sorts. The reserved resources in question could be provided by the cluster manager as the following:

- Limited processes (e.g., in some basic use cases of YARN), in which processes have their resource consumption metered but are not prevented from accessing each other’s resource by default.
- *Containers* (e.g., in the case of Mesos or Kubernetes), in which containers are a relatively lightweight resource reservation technology that is born out of the cgroups and namespaces of the Linux kernel and have known their most popular iteration with the Docker project.
- They also could be one of the above deployed on *virtual machines* (VMs), themselves coming with specific cores and memory reservation.

CLUSTER OPERATIONS

Detailing the different levels of isolations entailed by these three techniques is beyond the scope of this book but well worth exploring for production setups.

Note that in an enterprise-level production cluster management domain, we also encounter notions such as job queues, priorities, multitenancy options, and preemptions that are properly the domain of that cluster manager and therefore not something that is very frequently talked about in material that is focused on Spark.

However, it will be essential for you to have a firm grasp of the specifics of your cluster manager setup to understand how to be a “good citizen” on a cluster of machines, which are often shared by several teams. There are many good practices on how to run a proper cluster manager while many teams compete for its resources. And for those recommendations, you should consult both the references listed at the end of this chapter and your local DevOps team.

Understanding Resilience and Fault Tolerance in a Distributed System

Resilience and fault tolerance are absolutely essential for a distributed application: they are the condition by which we will be able to perform the user’s computation to completion. Nowadays, clusters are made of commodity machines that are ideally operated near peak capacity over their lifetime.

To put it mildly, hardware breaks quite often. A *resilient* application can make progress with its process despite latencies and noncritical faults in its distributed environment. A *fault-tolerant* application is able to succeed and complete its process despite the unplanned termination of one or several of its nodes.

This sort of resiliency is especially relevant in stream processing given that the applications we're scheduling are supposed to live for an undetermined amount of time. That undetermined amount of time is often correlated with the life cycle of the data source. For example, if we are running a retail website and we are analyzing transactions and website interactions as they come into the system against the actions and clicks and navigation of users visiting the site, we potentially have a data source that will be available for the entire duration of the lifetime of our business, which we hope to be very long, if our business is going to be successful.

As a consequence, a system that will process our data in a streaming fashion should run uninterrupted for long periods of time.

This “show must go on” approach of streaming computation makes the resiliency and fault-tolerance characteristics of our applications more important. For a batch job, we could launch it, hope it would succeed, and relaunch if we needed to change it or in case of failure. For an online streaming Spark pipeline, this is not a reasonable assumption.

Fault Recovery

In the context of fault tolerance, we are also interested in understanding how long it takes to recover from failure of one particular node. Indeed, stream processing has a particular aspect: data continues being generated by the data source in real time. To deal with a batch computing failure, we always have the opportunity to restart from scratch and accept that obtaining the results of

computation will take longer. Thus, a very primitive form of fault tolerance is detecting the failure of a particular node of our deployment, stopping the computation, and restarting from scratch. That process can take more than twice the original duration that we had budgeted for that computation, but if we are not in a hurry, this is still acceptable.

For stream processing, *we need to keep receiving data* and thus potentially storing it, if the recovering cluster is not ready to assume any processing yet. This can pose a problem at a high throughput: if we try restarting from scratch, we will need not only to reprocess all of the data that we have observed since the beginning of the application—which in and of itself can be a challenge—but during that reprocessing of historical data, we will need it to continue receiving and thus potentially storing new data that was generated while we were trying to catch up. This pattern of restarting from scratch is something so intractable for streaming that we will pay special attention to Spark’s ability to restart only *minimal* amounts of computation in the case that a node becomes unavailable or nonfunctional.

Cluster Manager Support for Fault Tolerance

We want to highlight why it is still important to understand Spark’s fault tolerance guarantees, even if there are similar features present in the cluster managers of YARN, Mesos, or Kubernetes. To understand this, we can consider that cluster managers help with fault tolerance when they work hand in hand with a framework that is able to report

failures and request new resources to cope with those exceptions. Spark possesses such capabilities.

For example, *production* cluster managers such as YARN, Mesos, or Kubernetes have the ability to detect a node's failure by inspecting endpoints on the node and asking the node to report on its own readiness and liveness state. If these cluster managers detect a failure and they have spare capacity, they will replace that node with another, made available to Spark. That particular action implies that the Spark executor code will start anew in another node, and then attempt to join the existing Spark cluster.

The cluster manager, by definition, does not have introspection capabilities into the applications being run on the nodes that it reserves. Its responsibility is limited to the container that runs the user's code.

That responsibility boundary is where the Spark resilience features start. To recover from a failed node, Spark needs to do the following:

- Determine whether that node contains some state that should be reproduced in the form of checkpointed files
- Understand at which stage of the job a node should rejoin the computation

The goal here is for us to explore that if a node is being replaced by the cluster manager, Spark has capabilities that allow it to take advantage of this new node and to distribute computation onto it.

Within this context, our focus is on Spark's responsibilities as an application and underline the capabilities of a cluster manager only when necessary: for instance, a node could be replaced because of a hardware failure or because its work was simply preempted by a higher-priority job. Apache Spark is blissfully unaware of the *why*, and focuses on the *how*.

Data Delivery Semantics

As you have seen in the streaming model, the fact that streaming jobs act on the basis of data that is generated in real time means that intermediate results need to be provided to the *consumer* of that streaming pipeline on a regular basis.

Those results are being produced by some part of our cluster. Ideally, we would like those observable results to be coherent, in line, and in real time with respect to the arrival of data. This means that we want results that are exact, and we want them as soon as possible.

However, distributed computation has its own challenges in that it sometimes includes not only individual nodes failing, as we have mentioned, but it also encounters situations like *network partitions*, in which some parts of our cluster are not able to communicate with other parts of that cluster, as illustrated in Figure 5-1.

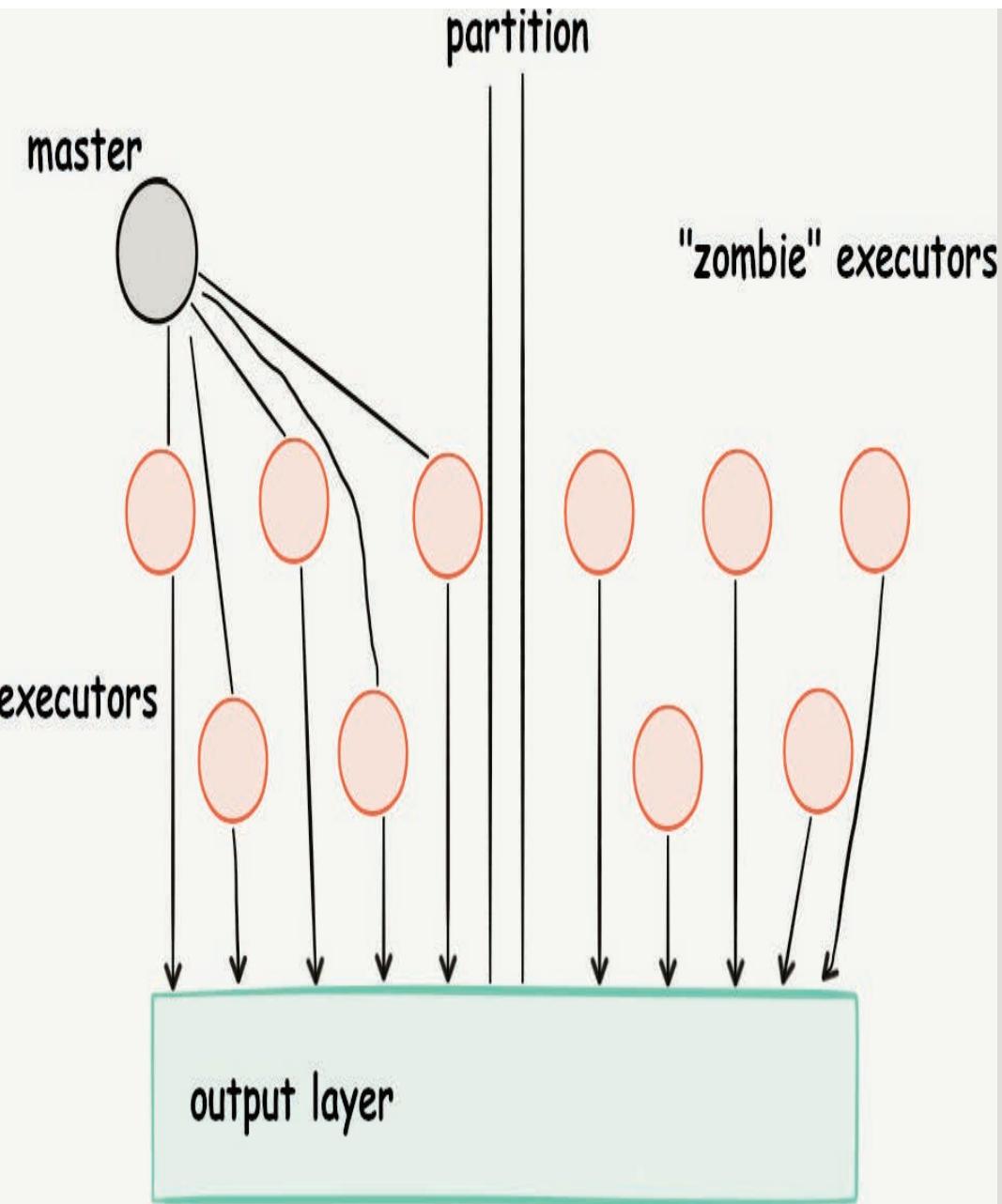


Figure 5-1. A network partition

Spark has been designed using a *driver/executor* architecture. A specific machine, the *driver*, is tasked with keeping track of the *job progression* along with the job submissions of a user, and the computation of that program occurs as the data arrives. However, if the network partitions separate some part of the cluster, the *driver*

might be able to keep track of only the part of the executors that form the initial cluster. In the other section of our partition, we will find nodes that are entirely able to function, but will simply be unable to account for the proceedings of their computation to the *driver*.

This creates an interesting case in which those “zombie” nodes do not receive new tasks, but might well be in the process of completing some fragment of computation that they were previously given. Being unaware of the partition, they will report their results as any executor would. And because this reporting of results sometimes does not go through the *driver* (for fear of making the *driver* a bottleneck), the reporting of these zombie results could succeed.

Because the *driver*, a single point of bookkeeping, does not know that those zombie executors are still functioning and reporting results, it will reschedule the same tasks that the lost executors had to accomplish on new nodes. This creates a *double-answering* problem in which the zombie machines lost through partitioning and the machines bearing the rescheduled tasks both report the same results. This bears real consequences: one example of stream computation that we previously mentioned is routing tasks for financial transactions. A double withdrawal, in that context, or double stock purchase orders, could have tremendous consequences.

It is not only the aforementioned problem that causes different processing semantics. Another important reason is that when output from a stream-processing application and state checkpointing cannot be completed in one atomic operation, it will cause data corruption if failure happens between checkpointing and outputting.

These challenges have therefore led to a distinction between *at least once* processing and *at most once* processing:

At least once

This processing ensures that every element of a stream has been processed once or more.

At most once

This processing ensures that every element of the stream is processed once or less.

Exactly once

This is the combination of “at least once” and “at most once.”

At-least-once processing is the notion that we want to make sure that every chunk of initial data has been dealt with—it deals with the node failure we were talking about earlier. As we’ve mentioned, when a streaming process suffers a partial failure in which some nodes need to be replaced or some data needs to be recomputed, we need to reprocess the lost units of computation while keeping the ingestion of data going. That requirement means that if you do not respect at-least-once processing, there is a chance for you, under certain conditions, to lose data.

The antisymmetric notion is called at-most-once processing. At-most-once processing systems guarantee that the zombie nodes repeating the same results as a rescheduled node are treated in a coherent manner, in which we keep track of only one set of results. By keeping track of *what data* their results *were about*, we’re able to make sure we can discard repeated results, yielding at-most-once processing

guarantees. The way in which we achieve this relies on the notion of *idempotence* applied to the “last mile” of result reception.

Idempotence qualifies a function such that if we apply it twice (or more) to any data, we will get the same result as the first time. This can be achieved by keeping track of the data that we are reporting a result for, and having a bookkeeping system at the output of our streaming process.

Microbatching and One-Element-at-a-Time

In this section, we want to address two important approaches to stream processing: *bulk-synchronous processing*, and *one-at-a-time record processing*.

The objective of this is to connect those two ideas to the two APIs that Spark possesses for stream processing: Spark Streaming and Structured Streaming.

Microbatching: An Application of Bulk-Synchronous Processing

Spark Streaming, the more mature model of stream processing in Spark, is roughly approximated by what’s called a *Bulk Synchronous Parallelism* (BSP) system.

The gist of BSP is that it includes two things:

- A split distribution of asynchronous work

- A synchronous barrier, coming in at fixed intervals

The split is the idea that each of the successive steps of work to be done in streaming is separated in a number of parallel chunks that are roughly proportional to the number of executors available to perform this task. Each executor receives its own chunk (or chunks) of work and works separately until the second element comes in. A particular resource is tasked with keeping track of the progress of computation. With Spark Streaming, this is a synchronization point at the “driver” that allows the work to progress to the next step. Between those scheduled steps, all of the executors on the cluster are doing the same thing.

Note that what is being passed around in this scheduling process are the functions that describe the processing that the user wants to execute on the data. The data is already on the various executors, most often being delivered directly to these resources over the lifetime of the cluster.

This was coined “function-passing style” by Heather Miller in 2016 (and formalized in [\[Miller2016\]](#)): asynchronously pass safe functions to distributed, stationary, immutable data in a stateless container, and use lazy combinators to eliminate intermediate data structures.

The frequency at which further rounds of data processing are scheduled is dictated by a time interval. This time interval is an arbitrary duration that is measured in batch processing time; that is, what you would expect to see as a “wall clock” time observation in your cluster.

For stream processing, we choose to implement barriers at small, fixed intervals that better approximate the real-time notion of data processing.

BULK-SYNCHRONOUS PARALLELISM

BSP is a very generic model for thinking about parallel processing, introduced by Leslie Valiant in the 1990s. Intended as an abstract (mental) model above all else, it was meant to provide a pendant to the Von Neumann model of computation for parallel processing.

It introduces three key concepts:

- A number of components, each performing processing and/or memory functions
- A router that delivers messages point to point between components
- Facilities for synchronizing all or a subset of the component at a regular time interval L , where L is the periodicity parameter

The purpose of the bulk-synchronous model is to give clear definitions that allow thinking of the moments when a computation can be performed by agents each acting separately, while pooling their knowledge together on a regular basis to obtain one single, aggregate result. Valiant introduces the notion:

A computation consists of a sequence of supersteps. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep.

This model also proceeds to give guarantees about the scalability and cost of this mode of computation. (To learn more about this, consult [Valiant1990].) It was influential in the design of modern graph processing systems such as Google's Pregel. Here, we use it as a way to speak of the timing of synchronization of parallel computation in Spark's DStreams.

One-Record-at-a-Time Processing

By contrast, one-record-at-a-time processing functions by *pipelining*: it analyzes the whole computation as described by user-specified functions and deploys it as pipelines using the resources of the cluster. Then, the only remaining matter is to flow data through the various resources, following the prescribed pipeline. Note that in this

latter case, each step of the computation is materialized at some place in the cluster at any given point.

Systems that function mostly according to this paradigm include Apache Flink, Naiad, Storm, and IBM Streams. (You can read more on these in Chapter 29.) This does not necessarily mean that those systems are incapable of microbatching, but rather characterizes their major or most native mode of operation and makes a statement on their dependency on the process of pipelining, often at the heart of their processing.

The minimum latency, or time needed for the system to react to the arrival of one particular event, is very different between those two: minimum latency of the microbatching system is therefore the time needed to complete the reception of the current microbatch (the batch interval) plus the time needed to start a task at the executor where this data falls (also called scheduling time). On the other hand, a system processing records one by one can react as soon as it meets the event of interest.

Microbatching Versus One-at-a-Time: The Trade-Offs

Despite their higher latency, microbatching systems offer significant advantages:

- They are able to *adapt* at the synchronization barrier boundaries. That adaptation might represent the task of recovering from failure, if a number of executors have been shown to become deficient or lose data. The periodic

synchronization can also give us an opportunity to add or remove executor nodes, giving us the possibility to grow or shrink our resources depending on what we're seeing as the cluster load, observed through the throughput on the data source.

- Our BSP systems can sometimes have an easier time providing *strong consistency* because their batch determinations—that indicate the beginning and the end of a particular batch of data—are deterministic and recorded. Thus, any kind of computation can be redone and produce the same results the second time.
- Having data available *as a set* that we can probe or inspect at the beginning of the microbatch allows us to perform efficient optimizations that can provide ideas on the way to compute on the data. Exploiting that on *each* microbatch, we can consider the specific case rather than the general processing, which is used for all possible input. For example, we could take a sample or compute a statistical measure before deciding to process or drop each microbatch.

More importantly, the simple presence of the microbatch as a well-identified element also allows an efficient way of specifying programming for both batch processing (where the data is at rest and has been saved somewhere) and streaming (where the data is in flight). The microbatch, even for mere instants, *looks* like data at rest.

Bringing Microbatch and One-Record-at-a-Time Closer Together

The marriage between microbatching and one-record-at-a-time processing as is implemented in systems like Apache Flink or Naiad

is still a subject of research.¹.]

Although it does not solve every issue, Structured Streaming, which is backed by a main implementation that relies on microbatching, does not expose that choice at the API level, allowing for an evolution that is independent of a fixed-batch interval. In fact, the default internal execution model of Structured Streaming is that of microbatching with a dynamic batch interval. Structured Streaming is also implementing continuous processing for some operators, which is something we touch upon in Chapter 15.

Dynamic Batch Interval

What is this notion of *dynamic batch interval*? The dynamic batch interval is the notion that the recomputation of data in a streaming DataFrame or Dataset consists of an update of existing data with the new elements seen over the wire. This update is occurring based on a trigger and the usual basis of this would be time duration. That time duration is still determined based on a fixed world clock signal that we expect to be synchronized within our entire cluster and that represents a single synchronous source of time that is shared among every executor.

However, this trigger can also be the statement of “as often as possible.” That statement is simply the idea that a new batch should be started as soon as the previous one has been processed, given a reasonable initial duration for the first batch. This means that the system will launch batches as often as possible. In this situation, the latency that can be observed is closer to that of one-element-at-a-time

processing. The idea here is that the microbatches produced by this system will converge to the smallest manageable size, making our stream flow faster through the executor computations that are necessary to produce a result. As soon as that result is produced, a new query will be started and scheduled by the Spark driver.

Structured Streaming Processing Model

The main steps in Structured Streaming processing are as follows:

1. When the Spark driver triggers a new batch, processing starts with updating the account of data read from a data source, in particular, getting data offsets for the beginning and the end of the latest batch.
2. This is followed by logical planning, the construction of successive steps to be executed on data, followed by query planning (intrastep optimization).
3. And then the launch and scheduling of the actual computation by adding a new batch of data to update the continuous query that we're trying to refresh.

Hence, from the point of view of the computation model, we will see that the API is significantly different from Spark Streaming.

The Disappearance of the Batch Interval

We now briefly explain what Structured Streaming batches mean and their impact with respect to operations.

In Structured Streaming, the batch interval that we are using is no longer a computation budget. With Spark Streaming, the idea was

that if we produce data every two minutes and flow data into Spark’s memory every two minutes, we should produce the results of computation on that batch of data in at least two minutes, to clear the memory from our cluster for the next microbatch. Ideally, as much data flows out as flows in, and the usage of the collective memory of our cluster remains stable.

With Structured Streaming, without this fixed time synchronization, our ability to see performance issues in our cluster is more complex: a cluster that is unstable—that is, unable to “clear out” data by finishing to compute on it as fast as new data flows in—will see ever-growing batch processing times, with an accelerating growth. We can expect that keeping a hand on this batch processing time will be pivotal.

However, if we have a cluster that is correctly sized with respect to the throughput of our data, there are a lot of advantages to have an as-often-as-possible update. In particular, we should expect to see very frequent results from our Structured Streaming cluster with a higher granularity than we used to in the time of a conservative batch interval.

¹ One interesting Spark-related project that recently came out of the University of Berkeley is called Drizzle and uses “group scheduling” to form a sort of longer-lived pipeline that persists across several batches, for the purpose of creating near-continuous queries. See [Venkataraman2016]

Chapter 6. Spark's Resilience Model

In most cases, a streaming job is a long-running job. By definition, streams of data observed and processed over time lead to jobs that run continuously. As they process data, they might accumulate intermediary results that are difficult to reproduce after the data has left the processing system. Therefore, the cost of failure is considerable and, in some cases, complete recovery is intractable.

In distributed systems, especially those relying on commodity hardware, failure is a function of size: the larger the system, the higher the probability that some component fails at any time. Distributed stream processors need to factor this chance of failure in their operational model.

In this chapter, we look at the resilience that the Apache Spark platform provides us: how it's able to recover partial failure and what kinds of guarantees we are given for the data passing through the system when a failure occurs. We begin by getting an overview of the different internal components of Spark and their relation to the core data structure. With this knowledge, you can proceed to understand the impact of failure at the different levels and the measures that Spark offers to recover from such failure.

Resilient Distributed Datasets in Spark

Spark builds its data representations on *Resilient Distributed Datasets* (RDDs). Introduced in 2011 by the paper “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” [Zaharia2011], RDDs are the foundational data structure in Spark. It is at this ground level that the strong fault tolerance guarantees of Spark start.

RDDs are composed of partitions, which are segments of data stored on individual nodes and tracked by the Spark driver that is presented as a location-transparent data structure to the user.

We illustrate these components in Figure 6-1 in which the classic *word count* application is broken down into the different elements that comprise an RDD.

```
.textFile("...") .flatMap(l => l.split(" ")) .map(w => (w,1)) .reduceByKey(_ + _)
```

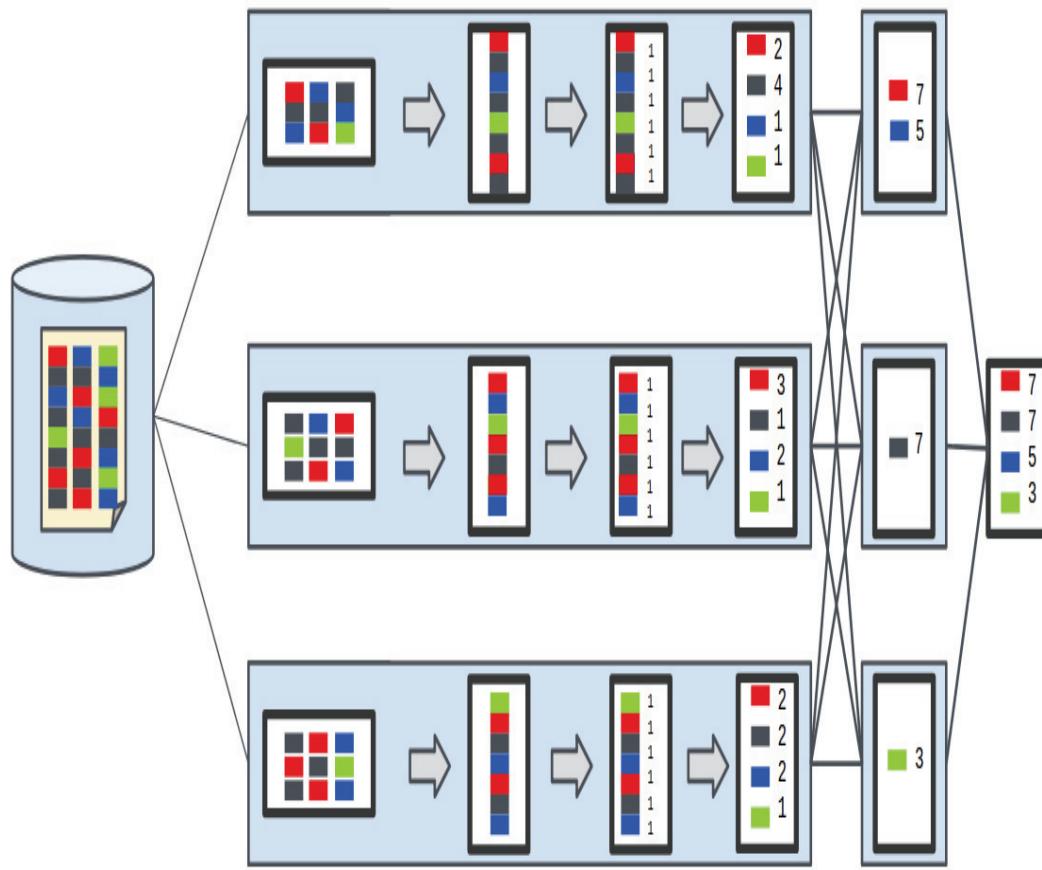


Figure 6-1. An RDD operation represented in a distributed system

The colored blocks are data elements, originally stored in a distributed filesystem, represented on the far left of the figure. The data is stored as partitions, illustrated as columns of colored blocks inside the file. Each partition is read into an executor, which we see as the horizontal blocks. The actual data processing happens within the executor. There, the data is transformed following the transformations described at the RDD level:

- `.flatMap(l => l.split(" "))` separates sentences into words separated by space.
- `.map(w => (w,1))` transforms each word into a tuple of the form (`<word>, 1`) in this way preparing the words for

counting.

- `.reduceByKey(_ + _)` computes the count, using the `<word>` as a key and applying a sum operation to the attached number.
- The final result is attained by bringing the partial results together using the same `reduce` operation.

RDDs constitute the programmatic core of Spark. All other abstractions, batch and streaming alike, including `DataFrames`, `DataSets`, and `DStreams` are built using the facilities created by RDDs, and, more important, they inherit the same fault tolerance capabilities. We provide a brief introduction of the RDDs programming model in “[RDDs as the Underlying Abstraction for DStreams](#)”.

Another important characteristic of RDDs is that Spark will try to keep their data preferably in-memory for as long as it is required and provided enough capacity in the system. This behavior is configurable through storage levels and can be explicitly controlled by calling caching operations.

We mention those structures here to present the idea that Spark tracks the progress of the user’s computation through modifications of the data. Indeed, knowing how far along we are in what the user wants to do through inspecting the control flow of his program (including loops and potential recursive calls) can be a daunting and error-prone task. It is much more reliable to define types of distributed data collections, and let the user create one from another, or from other data sources.

In [Figure 6-2](#), we show the same *word count* program, now in the form of the user-provided code (left) and the resulting internal RDD chain of operations. This dependency chain forms a particular kind of graph, a

Directed Acyclic Graph (DAG). The DAG informs the scheduler, appropriately called `DAGScheduler`, on how to distribute the computation and is also the foundation of the failure-recovery functionality, because it represents the internal data and their dependencies.

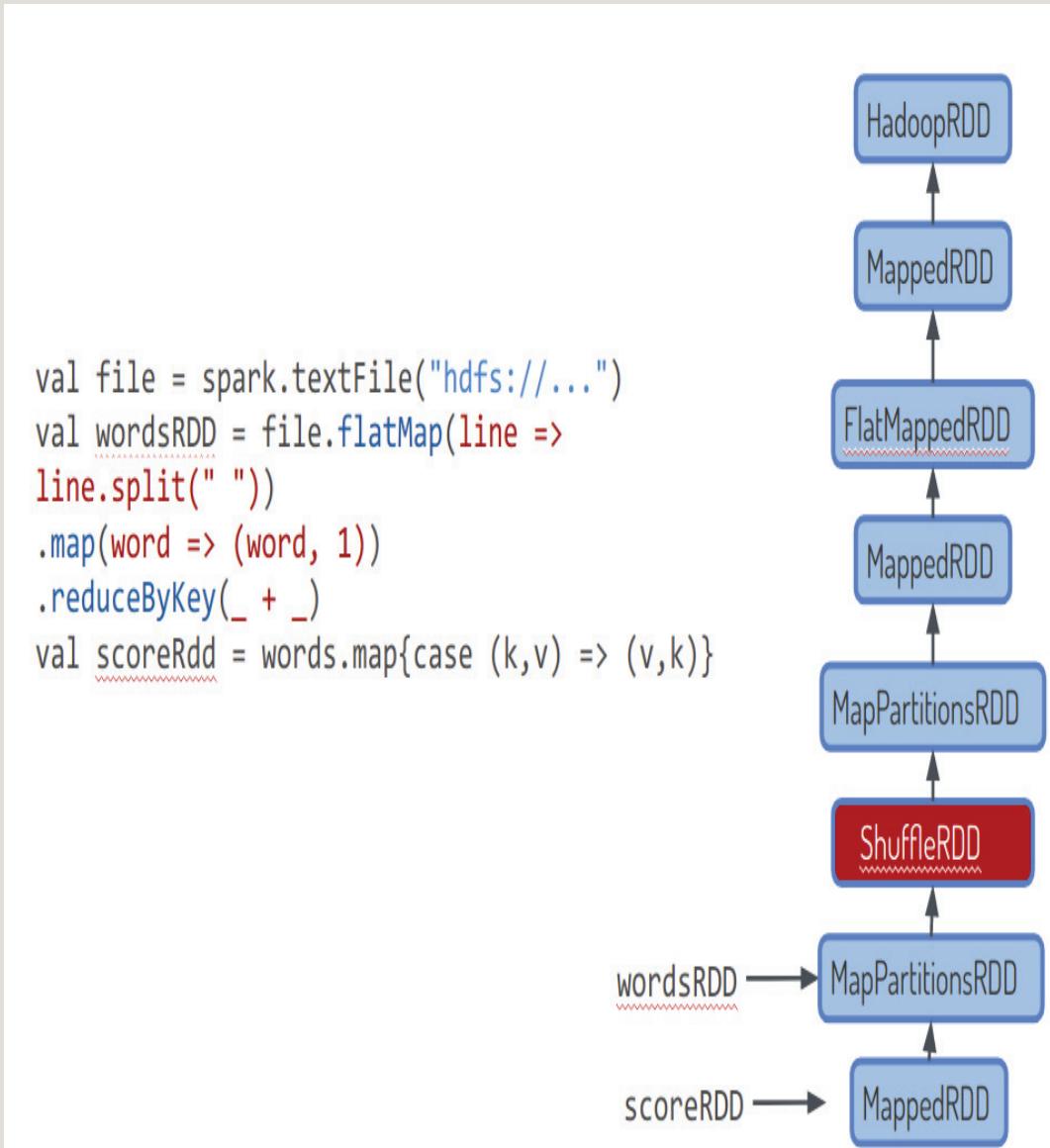


Figure 6-2. RDD lineage

As the system tracks the ordered creation of these distributed data collections, it tracks the work done, and what's left to accomplish.

Spark Components

To understand at what level fault tolerance operates in Spark, it's useful to go through an overview of the nomenclature of some core concepts. We begin by assuming that the user provides a program that ends up being divided into chunks and executed on various machines, as we saw in the previous section, and as depicted in Figure 6-3.

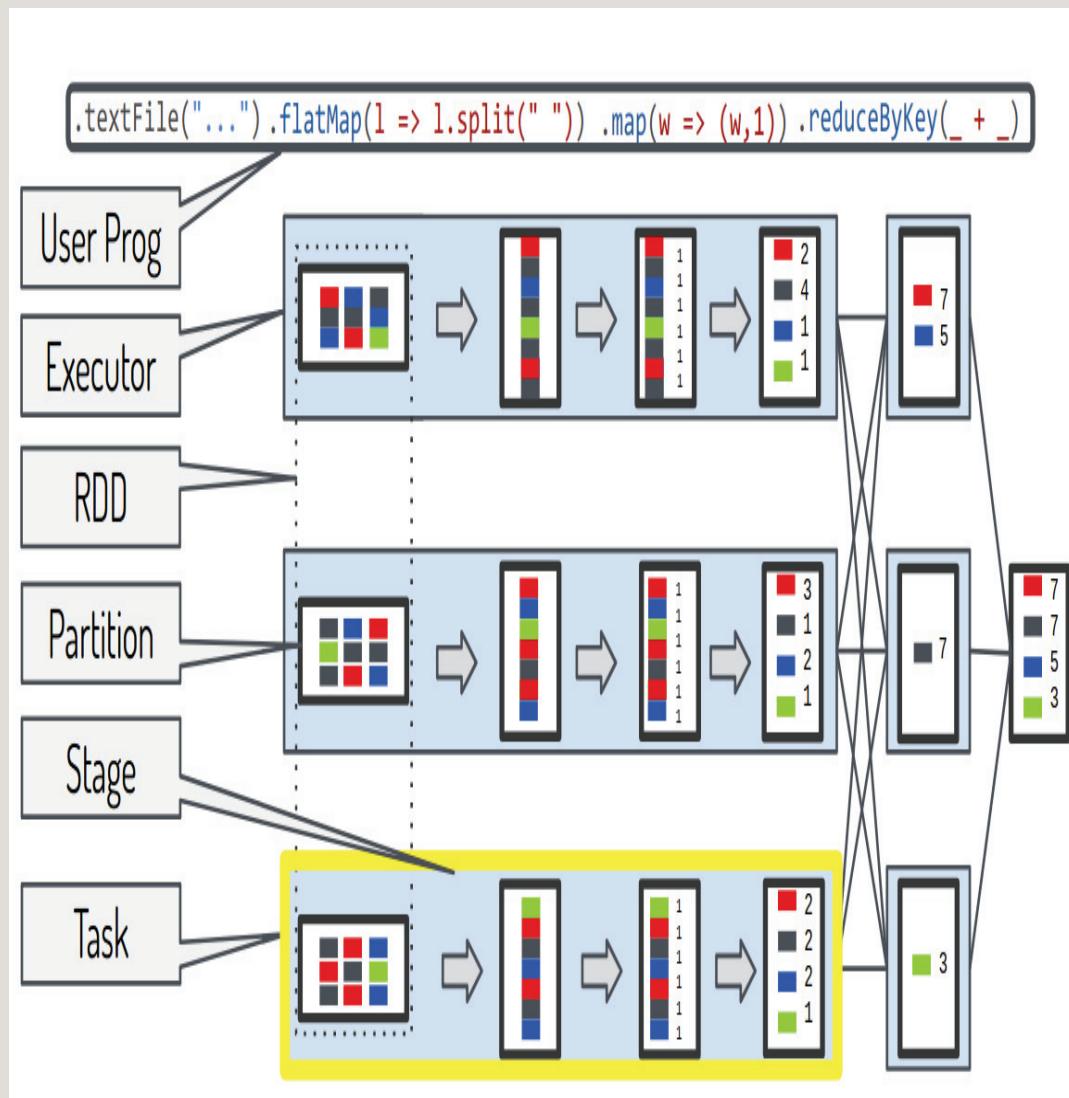


Figure 6-3. Spark nomenclature

Let's run down those steps, illustrated in Figure 6-3, which define the vocabulary of the Spark runtime:

User Program

The user application in Spark Streaming is composed of user-specified *function calls* operating on a resilient data structure (RDD, DStream, streaming DataSet, and so on), categorized as *actions* and *transformations*.

Transformed User Program

The user program may undergo adjustments that modify some of the specified calls to make them simpler, the most approachable and understandable of which is map-fusion.¹ Query plan is a similar but more advanced concept in Spark SQL.

RDD

A logical representation of a distributed, resilient, dataset. In the illustration, we see that the initial RDD comprises three parts, called partitions.

Partition

A partition is a physical segment of a dataset that can be loaded independently.

Stages

The user's operations are then grouped into *stages*, whose boundary separates user operations into steps that must be executed separately. For example, operations that require a shuffle of data across multiple nodes, such as a join between the results of two distinct upstream operations, mark a distinct stage. Stages in Apache Spark are the unit of sequencing: they are executed one after the other. At most one of any interdependent stages can be running at any given time.

Jobs

After these *stages* are defined, what internal actions Spark should take is clear. Indeed, at this stage, a set of interdependent *jobs* is defined. And jobs, precisely, are the vocabulary for a unit of scheduling. They describe the work at hand from the point of view of an entire Spark

cluster, whether it's waiting in a queue or currently being run across many machines. (Although it's not represented explicitly, in Figure 6-3, the job is the complete set of transformations.)

Tasks

Depending on where their source data is on the cluster, jobs can then be cut into *tasks*, crossing the conceptual boundary between distributed and single-machine computing: a task is a unit of local computation, the name for the local, executor-bound part of a job.

Spark aims to make sure that all of these steps are safe from harm and to recover quickly in the case of any incident occurring in any stage of this process. This concern is reflected in fault-tolerance facilities that are structured by the aforementioned notions: restart and checkpointing operations that occur at the task, job, stage, or program level.

Spark's Fault-Tolerance Guarantees

Now that we have seen the “pieces” that constitute the internal machinery in Spark, we are ready to understand that failure can happen at many different levels. In this section, we see Spark fault-tolerance guarantees organized by “increasing blast radius,” from the more modest to the larger failure. We are going to investigate the following:

- How Spark mitigates Task failure through restarts
- How Spark mitigates Stage failure through the shuffle service
- How Spark mitigates the disappearance of the *orchestrator* of the user program, through driver restarts

When you've completed this section, you will have a clear mental picture of the guarantees Spark affords us at runtime, letting you understand the failure scenarios that a well-configured Spark job can deal with.

Task Failure Recovery

Tasks can fail when the infrastructure on which they are running has a failure or logical conditions in the program lead to an sporadic job, like `OutOfMemory`, network, storage errors, or problems bound to the quality of the data being processed (e.g., a parsing error, a `NumberFormatException`, or a `NullPointerException` to name a few common exceptions).

If the input data of the task was stored, through a call to `cache()` or `persist()` and if the chosen storage level implies a replication of data (look for a storage level whose setting ends in `_2`, such as `MEMORY_ONLY_SER_2`), the task does not need to have its input recomputed, because a copy of it exists in complete form on another machine of the cluster. We can then use this input to restart the task. Table 6-1 summarizes the different storage levels configurable in Spark and their characteristics in terms of memory usage and replication factor.

Table 6-1. Spark storage levels

Level	Uses disk	Uses memory	Uses off-heap storage	Object (deserialized)	# of replicated copies
NONE					1
DISK_ONLY	X				1
DISK_ONLY_2	X				2
MEMORY_ONLY		X		X	1
MEMORY_ONLY_2		X		X	2
MEMORY_ONLY_SER		X			1
MEMORY_ONLY_SER_2		X			2
MEMORY_AND_DISK	X	X		X	1
MEMORY_AND_DISK_2	X	X		X	2
MEMORY_AND_DISK_SER	X	X			1
MEMORY_AND_DISK_SER_2	X	X			2
OFF_HEAP			X		1

If, however, there was no persistence or if the storage level does not guarantee the existence of a copy of the task's input data, the Spark driver will need to consult the DAG that stores the user-specified computation to determine which segments of the job need to be recomputed.

Consequently, without enough precautions to save either on the caching or on the storage level, the failure of a task can trigger the recomputation of several others, up to a stage boundary.

Stage boundaries imply a shuffle, and a shuffle implies that intermediate data will somehow be materialized: as we discussed, the shuffle transforms executors into data servers that can provide the data to any other executor serving as a destination.

As a consequence, these executors have a copy of the map operations that led up to the shuffle. Hence, executors that participated in a shuffle have a copy of the map operations that led up to it. But that's a lifesaver if you have a dying downstream executor, able to rely on the upstream servers of the shuffle (which serve the output of the map-like operation). What if it's the contrary: you need to face the crash of one of the upstream executors?

Stage Failure Recovery

We've seen that task failure (possibly due to executor crash) was the most frequent incident happening on a cluster and hence the most important event to mitigate. Recurrent task failures will lead to the failure of the stage that contains that task. This brings us to the second facility that allows Spark to resist arbitrary stage failures: the *shuffle service*.

When this failure occurs, it always means some rollback of the data, but a shuffle operation, by definition, depends on all of the prior executors involved in the step that precedes it.

As a consequence, since Spark 1.3 we have the shuffle service, which lets you work on map data that is saved and distributed through the cluster with a good locality, but, more important, through a server that is not a Spark task. It's an external file exchange service written in Java that has no dependency on Spark and is made to be a much longer-running service than a Spark executor. This additional service attaches as a separate process in all cluster modes of Spark and simply offers a data file

exchange for executors to transmit data reliably, right before a shuffle. It is highly optimized through the use of a netty backend, to allow a very low overhead in transmitting data. This way, an executor can shut down after the execution of its map task, as soon as the shuffle service has a copy of its data. And because data transfers are faster, this transfer time is also highly reduced, reducing the vulnerable time in which any executor could face an issue.

Driver Failure Recovery

Having seen how Spark recovers from the failure of a particular task and stage, we can now look at the facilities Spark offers to recover from the failure of the driver program. The driver in Spark has an essential role: it is the depository of the block manager, which knows where each block of data resides in the cluster. It is also the place where the DAG lives.

Finally, it is where the scheduling state of the job, its metadata, and logs resides. Hence, if the driver is lost, a Spark cluster as a whole might well have lost which stage it has reached in computation, what the computation actually consists of, and where the data that serves it can be found, in one fell swoop.

CLUSTER-MODE DEPLOYMENT

Spark has implemented what's called the *cluster deployment mode*, which allows the driver program to be hosted on the cluster, as opposed to the user's computer.

The deployment mode is one of two options: in client mode, the driver is launched in the same process as the client that submits the application. In cluster mode, however, the driver is launched from one of the worker processes inside the cluster, and the client process exits as soon as it

fulfills its responsibility of submitting the application without waiting for the application to finish.

This, in sum, allows Spark to operate an automatic driver restart, so that the user can start a job in a “fire and forget fashion,” starting the job and then closing their laptop to catch the next train. Every cluster mode of Spark offers a web UI that will let the user access the log of their application. Another advantage is that driver failure does not mark the end of the job, because the driver process will be relaunched by the cluster manager. But this only allows recovery from scratch, given that the temporary state of the computation—previously stored in the driver machine—might have been lost.

CHECKPOINTING

To avoid losing intermediate state in case of a driver crash, Spark offers the option of checkpointing; that is, recording periodically a snapshot of the application’s state to disk. The setting of the `sparkContext.setCheckpointDirectory()` option should point to reliable storage (e.g., Hadoop Distributed File System [HDFS]) because having the driver try to reconstruct the state of intermediate RDDs from its local filesystem makes no sense: those intermediate RDDs are being created on the executors of the cluster and should as such not require any interaction with the driver for backing them up.

We come back to the subject of checkpointing in detail much later, in Chapter 24. In the meantime, there is still one component of any Spark cluster whose potential failure we have not yet addressed: the master node.

Summary

This tour of Spark-core's fault tolerance and high-availability modes should have given you an idea of the main primitives and facilities offered by Spark and of their defaults. Note that none of this is so far specific to Spark Streaming or Structured Streaming, but that all these lessons apply to the streaming APIs in that they are required to deliver long-running, fault-tolerant and yet performant applications.

Note also that these facilities reflect different concerns in the frequency of faults for a particular cluster. These facilities reflect different concerns for the frequency of faults in a particular cluster:

- Features such as setting up a failover master node kept up-to-date through Zookeeper are really about avoiding a single point of failure in the design of a Spark application.
- The Spark Shuffle Service is here to avoid any problems with a shuffle step at the end of a long list of computation steps making the whole fragile through a faulty executor.

The later is a much more frequent occurrence. The first is about dealing with every possible condition, the second is more about ensuring smooth performance and efficient recovery.

¹ The process by which `l.map(foo).map(bar)` is changed into `l.map((x) => bar(foo(x)))`

Appendix A. References for Part I

- [Armbrust2018] Armbrust, M., T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark,” May 27, 2018. <https://stanford.io/2Jia3iY>.
- [Bhartia2016] Bhartia, R. “Optimize Spark-Streaming to Efficiently Process Amazon Kinesis Streams,” AWS Big Data blog, February 26, 2016. <https://amzn.to/2E7I69h>.
- [Chambers2018] Chambers, B., and Zaharia, M., *Spark: The Definitive Guide*. O’Reilly, 2018.
- [Chintapalli2015] Chintapalli, S., D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Musbaum, K. Patil, B. Peng, and P. Poulosky. “Benchmarking Streaming Computation Engines at Yahoo!” Yahoo! Engineering, December 18, 2015. <http://bit.ly/2bhgMJd>.
- [Das2013] Das, Tathagata. “Deep Dive With Spark Streaming,” Spark meetup, June 17, 2013. <http://bit.ly/2Q8Xzem>.
- [Das2014] Das, Tathagata, and Yuan Zhong. “Adaptive Stream Processing Using Dynamic Batch Sizing,” 2014 ACM Symposium on Cloud Computing. <http://bit.ly/2WTOuby>.
- [Das2015] Das, Tathagata. “Improved Fault Tolerance and Zero Data Loss in Spark Streaming,” Databricks

Engineering blog. January 15, 2015. <http://bit.ly/2HqH614>.

- [Dean2004] Dean, Jeff, and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters,” OSDI San Francisco, December, 2004. <http://bit.ly/15LeQej>.
- [Doley1987] Doley, D., C. Dwork, and L. Stockmeyer. “On the Minimal Synchronism Needed for Distributed Consensus,” *Journal of the ACM* 34(1) (1987): 77-97. <http://bit.ly/2LHRy9K>.
- [Dosa2007] Dósa, György. “The Tight Bound of First fit Decreasing Bin-Packing Algorithm Is $FFD(I) \leq (11/9)OPT(I) + 6/9$.” In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer-Verlag, 2007.
- [Dunner2016] Dünner, C., T. Parnell, K. Atasu, M. Sifalakis, and H. Pozidis. “High-Performance Distributed Machine Learning Using Apache Spark,” December 2016. <http://bit.ly/2JoSgH4>.
- [Fischer1985] Fischer, M. J., N. A. Lynch, and M. S. Paterson. “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM* 32(2) (1985): 374–382. <http://bit.ly/2Ee9tPb>.
- [Gibbons2004] Gibbons, J. “An unbounded spigot algorithm for the digits of π ,” *American Mathematical Monthly* 113(4) (2006): 318-328. <http://bit.ly/2VwwvH2>.
- [Greenberg2015] Greenberg, David. *Building Applications on Mesos*. O’Reilly, 2015.
- [Halevy2009] Halevy, Alon, Peter Norvig, and Fernando Pereira. “The Unreasonable Effectiveness of Data,” *IEEE*

Intelligent Systems (March/April 2009).

<http://bit.ly/2VCveD3>.

- [Karau2015] Karau, Holden, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly, 2015.
- [Kestelyn2015] Kestelyn, J. “Exactly-once Spark Streaming from Apache Kafka,” Cloudera Engineering blog, March 16, 2015. <http://bit.ly/2EniQfJ>.
- [Kleppmann2015] Kleppmann, Martin. “A Critique of the CAP Theorem,” arXiv.org:1509.05393, September 2015. <http://bit.ly/30jxsG4>.
- [Koeninger2015] Koeninger, Cody, Davies Liu, and Tathagata Das. “Improvements to Kafka Integration of Spark Streaming,” Databricks Engineering blog, March 30, 2015. <http://bit.ly/2Hn7dat>.
- [Kreps2014] Kreps, Jay. “Questioning the Lambda Architecture,” O'Reilly Radar, July 2, 2014. <https://oreil.ly/2LSEdqv>.
- [Lamport1998] Lamport, Leslie. “The Part-Time Parliament,” *ACM Transactions on Computer Systems* 16(2): 133–169. <http://bit.ly/2W3zr1R>.
- [Lin2010] Lin, Jimmy, and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool, 2010. <http://bit.ly/2YD9wMr>.
- [Lyon2013] Lyon, Brad F. “Musings on the Motivations for Map Reduce,” Nowhere Near Ithaca blog, June, 2013, <http://bit.ly/2Q3OHXe>.
- [Maas2014] Maas, Gérard. “Tuning Spark Streaming for Throughput,” Virdata Engineering blog, December 22, 2014.

[http://www.virdata.com/tuning-spark/.](http://www.virdata.com/tuning-spark/)

- [Marz2011] Marz, Nathan. “How to beat the CAP theorem,” Thoughts from the Red Planet blog, October 13, 2011. <http://bit.ly/2KpKDQq>.
- [Marz2015] Marz, Nathan, and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning, 2015.
- [Miller2016] Miller, H., P. Haller, N. Müller, and J. Boullier “Function Passing: A Model for Typed, Distributed Functional Programming,” *ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity, Onward!* November 2016: (82-97). <http://bit.ly/2EQASaf>.
- [Nasir2016] Nasir, M.A.U. “Fault Tolerance for Stream Processing Engines,” arXiv.org:1605.00928, May 2016. <http://bit.ly/2Mpz66f>.
- [Shapira2014] Shapira, Gwen. “Building The Lambda Architecture with Spark Streaming,” Cloudera Engineering blog, August 29, 2014. <http://bit.ly/2XoyHBS>.
- [Sharp2007] Sharp, Alexa Megan. “Incremental algorithms: solving problems in a changing world,” PhD diss., Cornell University, 2007. <http://bit.ly/2Ie8MGX>.
- [Valiant1990] Valiant, L.G. “Bulk-synchronous parallel computers,” *Communications of the ACM* 33:8 (August 1990). <http://bit.ly/2IgX3ar>.
- [Vavilapalli2013] Vavilapalli, et al. “Apache Hadoop YARN: Yet Another Resource Negotiator,” ACM Symposium on Cloud Computing, 2013. <http://bit.ly/2Xn3tuZ>.

- [Venkat2015] Venkat, B., P. Padmanabhan, A. Arokiasamy, and R. Uppalapati. “Can Spark Streaming survive Chaos Monkey?” The Netflix Tech Blog, March 11, 2015.
<http://bit.ly/2WkDJmr>.
- [Venkataraman2016] Venkataraman, S., P. Aurojit, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. “Drizzle: Fast and Adaptable Stream Processing at Scale,” Tech Report, UC Berkeley, 2016.
<http://bit.ly/2HW08Ot>.
- [White2010] White, Tom. *Hadoop: The Definitive Guide*, 4th ed. O’Reilly, 2015.
- [Zaharia2011] Zaharia, Matei, Mosharaf Chowdhury, et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” UCB/EECS-2011-82.
<http://bit.ly/2IfZE4q>.
- [Zaharia2012] Zaharia, Matei, Tathagata Das, et al. “Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing,” UCB/EECS-2012-259.
<http://bit.ly/2MpuY6c>.

Part II. Structured Streaming

In this part, we examine Structured Streaming.

We begin our journey by exploring a practical example that should help you build your intuition for the model. From there, we examine the API and get into the details of the following aspects of stream processing:

- Consuming data using sources
- Building data-processing logic using the rich Streaming Dataframe/Dataset API
- Understanding and working with event time
- Dealing with state in streaming applications
- Learning about arbitrary stateful transformations
- Writing the results to other systems using sinks

Before closing, we provide an overview of the operational aspects of Structured Streaming.

Finally, we explore the current developments in this exciting new streaming API and provide insights into experimental areas like machine learning applications and near-real-time data processing with continuous streaming.

Chapter 7. Introducing Structured Streaming

In data-intensive enterprises, we find many large datasets: log files from internet-facing servers, tables of shopping behavior, and NoSQL databases with sensor data, just to name a few examples. All of these datasets share the same fundamental life cycle: They started out empty at some point in time and were progressively filled by arriving data points that were directed to some form of secondary storage. This process of data arrival is nothing more than a *data stream* being materialized onto secondary storage. We can then apply our favorite analytics tools on those datasets *at rest*, using techniques known as *batch processing* because they take large chunks of data at once and usually take considerable amounts of time to complete, ranging from minutes to days.

The `Dataset` abstraction in *Spark SQL* is one such way of analyzing data at rest. It is particularly useful for data that is *structured* in nature; that is, it follows a defined schema. The `Dataset` API in Spark combines the expressivity of a SQL-like API with type-safe collection operations that are reminiscent of the Scala collections and the Resilient Distributed Dataset (RDD) programming model. At the same time, the `Dataframe` API, which is in nature similar to Python Pandas and R Dataframes, widens the audience of Spark users beyond the initial core of data

engineers who are used to developing in a functional paradigm. This higher level of abstraction is intended to support modern data engineering and data science practices by enabling a wider range of professionals to jump onto the big data analytics train using a familiar API.

What if, instead of having to wait for the data to "settle down," we could apply the same Dataset concepts to the data while it is in its original stream form?

The Structured Streaming model is an extension of the Dataset SQL-oriented model to handle data on the move:

- The data arrives from a *source* stream and is assumed to have a defined schema.
- The stream of events can be seen as rows that are appended to an unbounded table.
- To obtain results from the stream, we express our computation as queries over that table.
- By continuously applying the same query to the updating table, we create an output stream of processed events.
- The resulting events are offered to an output *sink*.
- The *sink* could be a storage system, another streaming backend, or an application ready to consume the processed data.

In this model, our theoretically *unbounded* table must be implemented in a physical system with defined resource constraints. Therefore, the implementation of the model requires certain

considerations and restrictions to deal with a potentially infinite data inflow.

To address these challenges, Structured Streaming introduces new concepts to the `Dataset` and `DataFrame` APIs, such as support for event time, *watermarking*, and different output modes that determine for how long past data is actually stored.

Conceptually, the Structured Streaming model blurs the line between batch and streaming processing, removing a lot of the burden of reasoning about analytics on fast-moving data.

First Steps with Structured Streaming

In the previous section, we learned about the high-level concepts that constitute Structured Streaming, such as sources, sinks, and queries. We are now going to explore Structured Streaming from a practical perspective, using a simplified web log analytics use case as an example.

Before we begin delving into our first streaming application, we are going to see how classical batch analysis in Apache Spark can be applied to the same use case.

This exercise has two main goals:

- First, most, if not all, streaming data analytics start by studying a static data sample. It is far easier to start a study with a file of data, gain intuition on how the data looks, what kind of patterns it shows, and define the process that we

require to extract the intended knowledge from that data. Typically, it's only after we have defined and tested our data analytics job, that we proceed to transform it into a streaming process that can apply our analytic logic to data on the move.

- Second, from a practical perspective, we can appreciate how Apache Spark simplifies many aspects of transitioning from a batch exploration to a streaming application through the use of uniform APIs for both batch and streaming analytics.

This exploration will allow us to compare and contrast the batch and streaming APIs in Spark and show us the necessary steps to move from one to the other.

ONLINE RESOURCES

For this example, we use Apache Web Server logs from the public 1995 NASA Apache web logs, originally from <http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html>.

For the purpose of this exercise, the original log file has been split into daily files and each log line has been formatted as JSON. The compressed NASA-weblogs file can be downloaded from <https://github.com/stream-processing-with-spark>.

Download this dataset and place it in a folder on your computer.

Batch Analytics

Given that we are working with archive log files, we have access to all of the data at once. Before we begin building our streaming

application, let's take a brief *intermezzo* to have a look at what a classical batch analytics job would look like.

ONLINE RESOURCES

For this example, we will use the `batch_weblogs` notebook in the online resources for the book, located at <https://github.com/stream-processing-with-spark>[<https://github.com/stream-processing-with-spark>].

First, we load the log files, encoded as JSON, from the directory where we unpacked them:

```
// This is the location of the unpackaged files. Update  
accordingly  
val logsDirectory = ???  
val rawLogs = sparkSession.read.json(logsDirectory)
```

Next, we declare the schema of the data as a `case class` to use the typed Dataset API. Following the formal description of the dataset (at [NASA-HTTP](#)), the log is structured as follows:

The logs are an ASCII file with one line per request, with the following columns:

- *Host making the request. A hostname when possible, otherwise the Internet address if the name could not be looked up.*
- *Timestamp in the format “DAY MON DD HH:MM:SS YYYY,” where DAY is the day of the week, MON is the name of the month, DD is the day of the month, HH:MM:SS is the time of day using a 24-hour clock, and YYYY is the year. The timezone is –0400.*
- *Request given in quotes.*
- *HTTP reply code.*
- *Bytes in the reply.*

Translating that schema to Scala, we have the following case class definition:

```
import java.sql.Timestamp
case class WebLog(host: String,
                  timestamp: Timestamp,
                  request: String,
                  http_reply: Int,
                  bytes: Long
                 )
```

NOTE

We use `java.sql.Timestamp` as the type for the timestamp because it's internally supported by Spark and does not require any additional cast that other options might require.

We convert the original JSON to a typed data structure using the previous schema definition:

```
import org.apache.spark.sql.functions._  
import org.apache.spark.sql.types.IntegerType  
// we need to narrow the `Interger` type because  
// the JSON representation is interpreted as `BigInteger`  
val preparedLogs = rawLogs.withColumn("http_reply",  
    $"http_reply".cast(IntegerType))  
val weblogs = preparedLogs.as[WebLog]
```

Now that we have the data in a structured format, we can begin asking the questions that interest us. As a first step, we would like to know how many records are contained in our dataset:

```
val recordCount = weblogs.count  
>recordCount: Long = 1871988
```

A common question would be: “what was the most popular URL per day?” To answer that, we first reduce the timestamp to the day of the month. We then group by this new dayOfMonth column and the request URL and we count over this aggregate. We finally order using descending order to get the top URLs first:

```
val topDailyURLs = weblogs.withColumn("dayOfMonth",  
    dayofmonth($"timestamp"))  
        .select($"request", "dayOfMonth")  
        .groupBy($"dayOfMonth",  
    $"request")  
  
.agg(count($"request").alias("count"))  
        .orderBy(desc("count"))  
  
topDailyURLs.show()  
+-----+-----+-----+  
|dayOfMonth|request|count|
```

13	GET /images/NASA-logosmall.gif	HTTP/1.0	12476
13	GET /htbin/cdt_main.pl	HTTP/1.0	7471
12	GET /images/NASA-logosmall.gif	HTTP/1.0	7143
13	GET /htbin/cdt_clock.pl	HTTP/1.0	6237
6	GET /images/NASA-logosmall.gif	HTTP/1.0	6112
5	GET /images/NASA-logosmall.gif	HTTP/1.0	5865
...			

Top hits are all images. What now? It's not unusual to see that the top URLs are images commonly used across a site. Our true interest lies in the content pages generating the most traffic. To find those, we first filter on `html` content and then proceed to apply the top aggregation we just learned.

As we can see, the request field is a quoted sequence of `[HTTP_VERB] URL [HTTP_VERSION]`. We will extract the URL and preserve only those ending in `.html`, `.htm`, or no extension (directories). This is a simplification for the purpose of this example:

```
val urlExtractor = """^GET (.+) HTTP/\d.\d""".r
val allowedExtensions = Set(".html", ".htm", "")
val contentPageLogs = weblogs.filter {log =>
  log.request match {
    case urlExtractor(url) =>
      val ext = url.takeRight(5).dropWhile(c => c != '.')
      allowedExtensions.contains(ext)
    case _ => false
  }
}
```

With this new dataset that contains only `.html`, `.htm`, and directories, we proceed to apply the same *top-k* function as earlier:

```
val topContentPages = contentPageLogs
  .withColumn("dayOfMonth", dayofmonth($"timestamp"))
```

```

.select($"request", $"dayOfMonth")
.groupBy($"dayOfMonth", $"request")
.agg(count($"request").alias("count"))
.orderBy(desc("count"))

topContentPages.show()
+-----+
+----+
| dayOfMonth |
| request | count |
+-----+
+----+
|      13 | GET /shuttle/countdown/liftoff.html HTTP/1.0"
| 4992 |      5 | GET /shuttle/countdown/ HTTP/1.0"
| 3412 |      6 | GET /shuttle/countdown/ HTTP/1.0"
| 3393 |      3 | GET /shuttle/countdown/ HTTP/1.0"
| 3378 |      13 | GET /shuttle/countdown/ HTTP/1.0"
| 3086 |      7 | GET /shuttle/countdown/ HTTP/1.0"
| 2935 |      4 | GET /shuttle/countdown/ HTTP/1.0"
| 2832 |      2 | GET /shuttle/countdown/ HTTP/1.0"
| 2330 |      ...

```

We can see that the most popular page that month was *liftoff.html*, corresponding to the coverage of the launch of the Discovery shuttle, as documented on the NASA archives. It's closely followed by *countdown/*, the days prior to the launch.

Streaming Analytics

In the previous section, we explored historical NASA web log records. We found trending events in those records, but much later than when the actual events happened.

One key driver for streaming analytics comes from the increasing demand of organizations to have timely information that can help them make decisions at many different levels.

We can use the lessons that we have learned while exploring the archived records using a batch-oriented approach and create a streaming job that will provide us with trending information as it happens.

The first difference that we observe with the batch analytics is the source of the data. For our streaming exercise, we will use a TCP server to simulate a web system that delivers its logs in real time. The simulator will use the same dataset but will feed it through a TCP socket connection that will embody the stream that we will be analyzing.

ONLINE RESOURCES

For this example, we will use the notebooks `weblog_TCP_server` and `streaming_weblogs` found in the online resources for the book, located at <https://github.com/stream-processing-with-spark>.

Connecting to a Stream

If you recall from the introduction of this chapter, Structured Streaming defines the concepts of sources and sinks as the key abstractions to consume a stream and produce a result. We are going to use the `TextSocketSource` implementation to connect to the server through a TCP socket. Socket connections are defined by the

host of the server and the port where it is listening for connections. These two configuration elements are required to create the socket source:

```
val stream = sparkSession.readStream
  .format("socket")
  .option("host", host)
  .option("port", port)
  .load()
```

Note how the creation of a stream is quite similar to the declaration of a static datasource in the batch case. Instead of using the `read` builder, we use the `readStream` construct and we pass to it the parameters required by the streaming source. As you will see during the course of this exercise and later on as we go into the details of Structured Streaming, the API is basically the same `DataFrame` and `Dataset` API for static data but with some modifications and limitations that you will learn in detail.

Preparing the Data in the Stream

The `socket` source produces a streaming `DataFrame` with one column, `value`, which contains the data received from the stream. See “[The Socket Source](#)” for additional details.

In the batch analytics case, we could load the data directly as JSON records. In the case of the `Socket` source, that data is plain text. To transform our raw data to `WebLog` records, we first require a schema. The schema provides the necessary information to parse the text to a

JSON object. It's the *structure* when we talk about *structured streaming*.

After defining a schema for our data, we proceed to create a Dataset, following these steps:

```
import java.sql.Timestamp
case class WebLog(host:String,
                  timestamp: Timestamp,
                  request: String,
                  http_reply:Int,
                  bytes: Long
)
val webLogSchema = Encoders.product[WebLog].schema ①
val jsonStream = stream.select(from_json($"value",
webLogSchema) as "record") ②
val webLogStream: Dataset[WebLog] =
jsonStream.select("record.*").as[WebLog] ③
```

- ① Obtain a schema from the `case class` definition
- ② Transform the text value to JSON using the JSON support built into Spark SQL
- ③ Use the `Dataset` API to transform the JSON records to `WebLog` objects

As a result of this process, we obtain a `Streaming Dataset` of `WebLog` records.

Operations on Streaming Dataset

The `webLogStream` we just obtained is of type `Dataset[WebLog]` like we had in the batch analytics job. The

difference between this instance and the batch version is that `webLogStream` is a streaming Dataset.

We can observe this by querying the object:

```
webLogStream.isStreaming  
> res: Boolean = true
```

At this point in the batch job, we were creating the first query on our data: How many records are contained in our dataset? This is a question that we can easily answer when we have access to all of the data. However, how do we count records that are constantly arriving? The answer is that some operations that we consider usual on a static Dataset, like counting all records, do not have a defined meaning on a Streaming Dataset.

As we can observe, attempting to execute the `count` query in the following code snippet will result in an `AnalysisException`:

```
val count = webLogStream.count()  
> org.apache.spark.sql.AnalysisException: Queries with  
streaming sources must  
be executed with writeStream.start();;
```

This means that the direct queries we used on a static Dataset or DataFrame now need two levels of interaction. First, we need to declare the transformations of our stream, and then we need to start the stream process.

Creating a Query

What are popular URLs? In what time frame? Now that we have immediate analytic access to the stream of web logs, we don't need to wait for a day or a month (or more than 20 years in the case of these NASA web logs) to have a rank of the popular URLs. We can have that information as trends unfold in much shorter windows of time.

First, to define the period of time of our interest, we create a window over some timestamp. An interesting feature of Structured Streaming is that we can define that time interval on the timestamp when the data was produced, also known as *event time*, as opposed to the time when the data is being processed.

Our window definition will be of five minutes of event data. Given that our timeline is simulated, the five minutes might happen much faster or slower than the clock time. In this way, we can clearly appreciate how Structured Streaming uses the timestamp information in the events to keep track of the event timeline.

As we learned from the batch analytics, we should extract the URLs and select only content pages, like *.html*, *.htm*, or directories. Let's apply that acquired knowledge first before proceeding to define our windowed query:

```
// A regex expression to extract the accessed URL from
weblog.request
val urlExtractor = """^GET (.+) HTTP/\d.\d""".r
val allowedExtensions = Set(".html", ".htm", "")

val contentPageLogs: String => Boolean = url => {
  val ext = url.takeRight(5).dropWhile(c => c != '.')
  allowedExtensions.contains(ext)
}
```

```
val urlWebLogStream = webLogStream.flatMap { weblog =>
    weblog.request match {
        case urlExtractor(url) if (contentPageLogs(url)) =>
            Some(weblog.copy(request = url))
        case _ => None
    }
}
```

We have converted the request to contain only the visited URL and filtered out all noncontent pages. Now, we define the windowed query to compute the top trending URLs:

```
val rankingURLStream = urlWebLogStream
    .groupBy($"request", window($"timestamp", "5 minutes",
    "1 minute"))
    .count()
```

Start the Stream Processing

All of the steps that we have followed so far have been to define the process that the stream will undergo. But no data has been processed yet.

To start a Structured Streaming job, we need to specify a `sink` and an `output mode`. These are two new concepts introduced by Structured Streaming:

- A `sink` defines where we want to materialize the resulting data; for example, to a file in a filesystem, to an in-memory table, or to another streaming system such as Kafka.
- The `output mode` defines how we want the results to be delivered: do we want to see all data every time, only updates, or just the new records?

These options are given to a `writeStream` operation. It creates the streaming query that starts the stream consumption, materializes the computations declared on the query, and produces the result to the output sink.

We visit all these concepts in detail later on. For now, let's use them empirically and observe the results.

For our query, shown in Example 7-1, we use the memory sink and output mode `complete` to have a fully updated table each time new records are added to the result of keeping track of the URL ranking.

Example 7-1. Writing a stream to a sink

```
val query = rankingURLStream.writeStream
  .queryName("urlranks")
  .outputMode("complete")
  .format("memory")
  .start()
```

The memory sink outputs the data to a temporary table of the same name given in the `queryName` option. We can observe this by querying the tables registered on Spark SQL:

```
scala> spark.sql("show tables").show()
+-----+-----+
|database|tableName|isTemporary|
+-----+-----+
|        | urlranks|      true|
+-----+-----+
```

In the expression in Example 7-1, `query` is of type `StreamingQuery` and it's a handler to control the query life cycle.

Exploring the Data

Given that we are accelerating the log timeline on the producer side, after a few seconds, we can execute the next command to see the result of the first windows, as illustrated in Figure 7-1.

Note how the processing time (a few seconds) is decoupled from the event time (hundreds of minutes of logs):

```
urlRanks.select($"request", $"window",
  $"count").orderBy(desc("count"))
```

request	window	count
"/shuttle/missions/sts-70/mission-sts-70.html"	{"start":"2018-02-02T18:18:00.000+01:00","end":"2018-02-02T18:23:00.000+01:00"}	8
"/shuttle/countdown"	{"start":"2018-02-02T18:17:00.000+01:00","end":"2018-02-02T18:22:00.000+01:00"}	8
"/shuttle/countdown"	{"start":"2018-02-02T18:18:00.000+01:00","end":"2018-02-02T18:23:00.000+01:00"}	8
"/shuttle/countdown"	{"start":"2018-02-02T18:20:00.000+01:00","end":"2018-02-02T18:25:00.000+01:00"}	7
"/shuttle/countdown"	{"start":"2018-02-02T18:21:00.000+01:00","end":"2018-02-02T18:26:00.000+01:00"}	7
"/shuttle/countdown/liftoff.html"	{"start":"2018-02-02T18:22:00.000+01:00","end":"2018-02-02T18:27:00.000+01:00"}	7
"/shuttle/missions/sts-70/mission-sts-70.html"	{"start":"2018-02-02T18:17:00.000+01:00","end":"2018-02-02T18:22:00.000+01:00"}	7
"/shuttle/countdown/liftoff.html"	{"start":"2018-02-02T18:20:00.000+01:00","end":"2018-02-02T18:25:00.000+01:00"}	6
"/shuttle/countdown"	{"start":"2018-02-02T18:16:00.000+01:00","end":"2018-02-02T18:21:00.000+01:00"}	6
"/ksc.html"	{"start":"2018-02-02T18:17:00.000+01:00","end":"2018-02-02T18:22:00.000+01:00"}	6

Figure 7-1. URL ranking: query results by window

We explore event time in detail in Chapter 12.

Summary

In these first steps into Structured Streaming, you have seen the process behind the development of a streaming application. By

starting with a batch version of the process, you gained intuition about the data, and using those insights, we created a streaming version of the job. In the process, you could appreciate how close the *structured* batch and the streaming APIs are, albeit we also observed that some usual batch operations do now apply in a streaming context.

With this exercise, we hope to have increased your curiosity about Structured Streaming. You're now ready for the learning path through this section.

Chapter 8. The Structured Streaming Programming Model

Structured Streaming builds on the foundations laid on top of the Spark SQL DataFrames and Datasets APIs of Spark SQL. By extending these APIs to support streaming workloads, Structured Streaming inherits the traits of the high-level language introduced by Spark SQL as well as the underlying optimizations, including the use of the Catalyst query optimizer and the low overhead memory management and code generation delivered by Project Tungsten. At the same time, Structured Streaming becomes available in all the supported language bindings for Spark SQL. These are: Scala, Java, Python, and R, although some of the advanced state management features are currently available only in Scala. Thanks to the intermediate query representation used in Spark SQL, the performance of the programs is identical regardless of the *language binding* used.

Structured Streaming introduces support for event time across all windowing and aggregation operations, making it easy to program logic that uses the time when events were generated, as opposed to the time when they enter the processing engine, also known as *processing time*. You learned these concepts in “The Effect of Time”.

With the availability of Structured Streaming in the Spark ecosystem, Spark manages to unify the development experience between *classic* batch and stream-based data processing.

In this chapter, we examine the programming model of Structured Streaming by following the sequence of steps that are usually required to create a streaming job with Structured Streaming:

- Initializing Spark
- Sources: acquiring streaming data
- Declaring the operations we want to apply to the streaming data
- Sinks: output the resulting data

Initializing Spark

Part of the visible unification of APIs in Spark is that `SparkSession` becomes the single entry point for *batch* and *streaming* applications that use Structured Streaming.

Therefore, our entry point to create a Spark job is the same as when using the Spark batch API: we instantiate a `SparkSession` as demonstrated in Example 8-1.

Example 8-1. Creating a local Spark Session

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("StreamProcessing")
```

```
.master("local[*]")
.getOrCreate()
```

USING THE SPARK SHELL

When using the Spark shell to explore Structured Streaming, the `SparkSession` is already provided as `spark`. We don't need to create any additional context to use Structured Streaming.

Sources: Acquiring Streaming Data

In Structured Streaming, a *source* is an abstraction that lets us consume data from a streaming data producer. Sources are not directly created. Instead, the `sparkSession` provides a builder method, `readStream`, that exposes the API to specify a streaming source, called a *format*, and provide its configuration.

For example, the code in [Example 8-2](#) creates a `File` streaming source. We specify the type of source using the `format` method. The method `schema` lets us provide a schema for the data stream, which is mandatory for certain source types, such as this `File` source.

Example 8-2. File streaming source

```
val fileStream = spark.readStream
  .format("json")
  .schema(schema)
  .option("mode", "DROPMALFORMED")
  .load("/tmp/datasrc")

>fileStream:
org.apache.spark.sql.DataFrame = [id: string, timestamp:
timestamp ... ]
```

Each source implementation has different options, and some have tunable parameters. In Example 8-2, we are setting the option mode to `DROPMALFORMED`. This option instructs the JSON stream processor to drop any line that neither complies with the JSON format nor matches the provided schema.

Behind the scenes, the call to `spark.readStream` creates a `DataStreamReader` instance. This instance is in charge of managing the different options provided through the builder method calls. Calling `load(...)` on this `DataStreamReader` instance validates the options provided to the builder and, if everything checks, it returns a streaming `DataFrame`.

NOTE

We can appreciate the symmetry in the Spark API: while `readStream` provides the options to declare the source of the stream, `writeStream` lets us specify the output sink and the output mode required by our process. They are the counterparts of `read` and `write` in the `DataFrame` APIs. As such, they provide an easy way to remember the execution mode used in a Spark program:

- `read/write`: Batch operation
- `readStream/writeStream`: Streaming operation

In our example, this streaming `DataFrame` represents the stream of data that will result from monitoring the provided *path* and processing each new file in that path as JSON-encoded data, parsed using the schema provided. All malformed code will be dropped from this data stream.

Loading a streaming source is lazy. What we get is a representation of the stream, embodied in the streaming DataFrame instance, that we can use to express the series of transformations that we want to apply to it in order to implement our specific business logic. Creating a streaming DataFrame does not result in any data actually being consumed or processed until the stream is materialized. This requires a *query*, as you will see further on.

Available Sources

As of Spark v2.4.0, the following streaming sources are supported:

`json, orc, parquet, csv, text, textFile`

These are all file-based streaming sources. The base functionality is to monitor a path (folder) in a filesystem and consume files atomically placed in it. The files found will then be parsed by the formatter specified. For example, if `json` is provided, the Spark `json` reader will be used to process the files, using the schema information provided.

`socket`

Establishes a client connection to a TCP server that is assumed to provide text data through a socket connection.

`kafka`

Creates Kafka consumer able to retrieve data from Kafka.

`rate`

Generates a stream of rows at the rate given by the `rowsPerSecond` option. It's mainly intended as a testing source.

We look at sources in detail in [Chapter 10](#).

Transforming Streaming Data

As we saw in the previous section, the result of calling `load` is a streaming DataFrame. After we have created our streaming DataFrame using a source, we can use the Dataset or DataFrame API to express the logic that we want to apply to the data in the stream in order to implement our specific use case.

WARNING

Remember that DataFrame is an alias for Dataset[Row]. Although this might seem like a small technical distinction, when used from a typed language such as Scala, the Dataset API presents a typed interface, whereas the DataFrame usage is untyped. When the structured API is used from a dynamic language such as Python, the DataFrame API is the only available API.

There's also a performance impact when using operations on a typed Dataset. Although the SQL expressions used by the DataFrame API can be understood and further optimized by the query planner, closures provided in Dataset operations are opaque to the query planner and therefore might run slower than the exact same DataFrame counterpart.

Assuming that we are using data from a sensor network, in [Example 8-3](#) we are selecting the fields `deviceId`, `timestamp`, `sensorType`, and `value` from a `sensorStream` and filtering to only those records where the sensor is of type `temperature` and its `value` is higher than the given threshold.

Example 8-3. Filter and projection

```
val highTempSensors = sensorStream
    .select($"deviceId", $"timestamp", $"sensorType", $"value")
    .where($"sensorType" === "temperature" && $"value" >
threshold)
```

Likewise, we can aggregate our data and apply operations to the groups over time. Example 8-4 shows that we can use `timestamp` information from the event itself to define a time window of five minutes that will slide every minute. We cover event time in detail in Chapter 12.

What is important to grasp here is that the Structured Streaming API is practically the same as the Dataset API for batch analytics, with some additional provisions specific to stream processing.

Example 8-4. Average by sensor type over time

```
val avgBySensorTypeOverTime = sensorStream
    .select($"timestamp", $"sensorType", $"value")
    .groupBy(window($"timestamp", "5 minutes", "1 minute"),
$"sensorType")
    .agg(avg($"value"))
```

If you are not familiar with the structured APIs of Spark, we suggest that you familiarize yourself with it. Covering this API in detail is beyond the scope of this book. We recommend *Spark: The Definitive Guide* (O'Reilly, 2018) by Bill Chambers and Matei Zaharia as a comprehensive reference.

Streaming API Restrictions on the DataFrame API

As we hinted in the previous chapter, some operations that are offered by the standard DataFrame and Dataset API do not make sense

on a streaming context.

We gave the example of `stream.count`, which does not make sense to use on a stream. In general, operations that require immediate materialization of the underlying dataset are not allowed.

These are the API operations not directly supported on streams:

- `count`
- `show`
- `describe`
- `limit`
- `take(n)`
- `distinct`
- `foreach`
- `sort`
- multiple stacked aggregations

Next to these operations, stream-stream and static-stream joins are partially supported.

UNDERSTANDING THE LIMITATIONS

Although some operations, like `count` or `limit`, do not make sense on a stream, some other stream operations are computationally difficult. For example, `distinct` is one of them. To filter duplicates in an arbitrary stream, it would require that you remember all of the data seen so far and compare each new record with all records

already seen. The first condition would require infinite memory and the second has a computational complexity of $O(n^2)$, which becomes prohibitive as the number of elements (n) increases.

OPERATIONS ON AGGREGATED STREAMS

Some of the unsupported operations become defined after we apply an aggregation function to the stream. Although we can't count the stream, we could count messages received per minute or count the number of devices of a certain type.

In Example 8-5, we define a count of events per `sensorType` per minute.

Example 8-5. Count of sensor types over time

```
val avgBySensorTypeOverTime = sensorStream
    .select($"timestamp", $"sensorType")
    .groupBy(window($"timestamp", "1 minutes", "1 minute"),
$"sensorType")
    .count()
```

Likewise, it's also possible to define a `sort` on aggregated data, although it's further restricted to queries with output mode `complete`. We examine about output modes in greater detail in “`outputMode`”.

STREAM DEDUPLICATION

We discussed that `distinct` on an arbitrary stream is computationally difficult to implement. But if we can define a key that informs us when an element in the stream has already been seen, we can use it to remove duplicates:

```
stream.dropDuplicates("key-column") ...
```

WORKAROUNDS

Although some operations are not supported in the exact same way as in the *batch* model, there are alternative ways to achieve the same functionality:

`foreach`

Although `foreach` cannot be directly used on a stream, there's a *foreach sink* that provides the same functionality.

Sinks are specified in the output definition of a stream.

`show`

Although `show` requires an immediate materialization of the query, and hence it's not possible on a streaming Dataset, we can use the `console` sink to output data to the screen.

Sinks: Output the Resulting Data

All operations that we have done so far—such as creating a stream and applying transformations on it—have been declarative. They define from where to consume the data and what operations we want to apply to it. But up to this point, there is still no data flowing through the system.

Before we can initiate our stream, we need to first define *where* and *how* we want the output data to go:

- *Where* relates to the streaming sink: the receiving side of our streaming data.
- *How* refers to the output mode: how to treat the resulting records in our stream.

From the API perspective, we materialize a stream by calling `writeStream` on a streaming DataFrame or Dataset, as shown in Example 8-6.

Calling `writeStream` on a streaming Dataset creates a `DataStreamWriter`. This is a builder instance that provides methods to configure the output behavior of our streaming process.

Example 8-6. File streaming sink

```
val query = stream.writeStream
  .format("json")
  .queryName("json-writer")
  .outputMode("append")
  .option("path", "/target/dir")
  .option("checkpointLocation", "/checkpoint/dir")
  .trigger(ProcessingTime("5 seconds"))
  .start()

>query: org.apache.spark.sql.streaming.StreamingQuery = ...
```

We cover sinks in detail in Chapter 11.

format

The `format` method lets us specify the output sink by providing the name of a built-in sink or the fully qualified name of a custom sink.

As of Spark v2.4.0, the following streaming sinks are available:

console sink

A sink that prints to the standard output. It shows a number of rows configurable with the option `numRows`.

file sink

File-based and format-specific sink that writes the results to a filesystem. The format is specified by providing the format name: `csv`, `hive`, `json`, `orc`, `parquet`, `avro`, or `text`.

kafka sink

A Kafka-specific producer sink that is able to write to one or more Kafka topics.

memory sink

Creates an in-memory table using the provided query name as table name. This table receives continuous updates with the results of the stream.

foreach sink

Provides a programmatic interface to access the stream contents, one element at the time.

foreachBatch sink

`foreachBatch` is a programmatic sink interface that provides access to the complete `DataFrame` that corresponds to each underlying microbatch of the Structured Streaming execution.

outputMode

The `outputMode` specifies the semantics of how records are added to the output of the streaming query. The supported modes are `append`, `update`, and `complete`:

append

(default mode) Adds only *final* records to the output stream. A record is considered *final* when no new records of the incoming stream can modify its value. This is always the case with linear transformations like those resulting from applying projection, filtering, and mapping. This mode guarantees that each resulting record will be output only once.

update

Adds new and updated records since the last trigger to the output stream. `update` is meaningful only in the context of an aggregation, where aggregated values change as new records arrive. If more than one incoming record changes a single result, all changes between trigger intervals are collated into one output record.

complete

`complete` mode outputs the complete internal representation of the stream. This mode also relates to aggregations, because for nonaggregated streams, we would need to remember all records seen so far, which is unrealistic. From a practical perspective, `complete` mode is recommended only when you are aggregating values over low-cardinality criteria, like *count of visitors by country*, for which we know that the number of countries is bounded.

UNDERSTANDING THE APPEND SEMANTIC

When the streaming query contains aggregations, the definition of final becomes nontrivial. In an aggregated computation, new incoming records might change an existing aggregated value when they comply with the aggregation criteria used. Following our definition, we cannot output a record using `append` until we know

that its value is final. Therefore, the use of the append output mode in combination with aggregate queries is restricted to queries for which the aggregation is expressed using event-time and it defines a watermark. In that case, append will output an event as soon as the watermark has expired and hence it's considered that no new records can alter the aggregated value. As a consequence, output events in append mode will be delayed by the aggregation time window plus the watermark offset.

queryName

With `queryName`, we can provide a name for the query that is used by some sinks and also presented in the job description in the Spark Console, as depicted in Figure 8-1.

Completed Jobs (9)					
Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8 (74af80d4-59c3-4012-bc3b-49d307b4a945)	json-writer id = 7318ab44-220e-4ef1-a5e6-07e0d3209337 runId = 74af80d4-59c3-4012-bc3b-49d307b4a945 batch = 8 start at <console>; 27	2018/02/25 19:51:30	20 ms	1/1	1/1
7 (74af80d4-59c3-4012-bc3b-49d307b4a945)	json-writer id = 7318ab44-220e-4ef1-a5e6-07e0d3209337 runId = 74af80d4-59c3-4012-bc3b-49d307b4a945 batch = 7 start at <console>; 27	2018/02/25 19:51:28	33 ms	1/1	1/1
6 (74af80d4-59c3-4012-bc3b-49d307b4a945)	json-writer id = 7318ab44-220e-4ef1-a5e6-07e0d3209337 runId = 74af80d4-59c3-4012-bc3b-49d307b4a945 batch = 6 start at <console>; 27	2018/02/25 19:51:26	17 ms	1/1	1/1

Figure 8-1. Completed Jobs in the Spark UI showing the query name in the job description

option

With the `option` method, we can provide specific key–value pairs of configuration to the stream, akin to the configuration of the source. Each sink can have specific configuration we can customize using this method.

We can add as many `.option(...)` calls as necessary to configure the sink.

options

`options` is an alternative to `option` that takes a `Map[String, String]` containing all the key–value configuration parameters that we want to set. This alternative is more friendly to an externalized configuration model, where we don't know *a priori* the settings to be passed to the sink's configuration.

trigger

The optional `trigger` option lets us specify the frequency at which we want the results to be produced. By default, Structured Streaming will process the input and produce a result as soon as possible. When a trigger is specified, output will be produced at each trigger interval.

`org.apache.spark.sql.streaming.Trigger` provides the following supported triggers:

`ProcessingTime(<interval>)`

Lets us specify a time interval that will dictate the frequency of the query results.

`Once()`

A particular Trigger that lets us execute a streaming job once. It is useful for testing and also to apply a defined streaming job as a single-shot batch operation.

`Continuous(<checkpoint-interval>)`

This trigger switches the execution engine to the experimental continuous engine for low-latency processing. The `checkpoint-interval` parameter indicates the frequency of the asynchronous checkpointing for data resilience. It should not be confused with the `batch interval` of the `ProcessingTime` trigger. We explore this new execution option in Chapter 15.

start()

To materialize the streaming computation, we need to start the streaming process. Finally, `start()` materializes the complete job description into a streaming computation and initiates the internal scheduling process that results in data being consumed from the source, processed, and produced to the sink. `start()` returns a `StreamingQuery` object, which is a handle to manage the individual life cycle of each query. This means that we can simultaneously start and stop multiple queries independently of one other within the same `sparkSession`.

Summary

After reading this chapter, you should have a good understanding of the Structured Streaming programming model and API. In this

chapter, you learned the following:

- Each streaming program starts by defining a source and what sources are currently available.
- We can reuse most of the familiar `Dataset` and `DataFrame` APIs for transforming the streaming data.
- Some common operations from the `batch` API do not make sense in streaming mode.
- Sinks are the configurable definition of the stream output.
- The relation between output modes and aggregation operations in the stream.
- All transformations are lazy and we need to start our stream to get data flowing through the system.

In the next chapter, you apply your newly acquired knowledge to create a comprehensive stream-processing program. After that, we will zoom into specific areas of the Structured Streaming API, such as event-time handing, window definitions, the concept of watermarks, and arbitrary state handling.

CHAPTER 1

Meet Kafka

Every enterprise is powered by data. We take information in, analyze it, manipulate it, and create more as output. Every application creates data, whether it is log messages, metrics, user activity, outgoing messages, or something else. Every byte of data has a story to tell, something of importance that will inform the next thing to be done. In order to know what that is, we need to get the data from where it is created to where it can be analyzed. We see this every day on websites like Amazon, where our clicks on items of interest to us are turned into recommendations that are shown to us a little later.

The faster we can do this, the more agile and responsive our organizations can be. The less effort we spend on moving data around, the more we can focus on the core business at hand. This is why the pipeline is a critical component in the data-driven enterprise. How we move the data becomes nearly as important as the data itself.

Any time scientists disagree, it's because we have insufficient data. Then we can agree on what kind of data to get; we get the data; and the data solves the problem. Either I'm right, or you're right, or we're both wrong. And we move on.

—Neil deGrasse Tyson

Publish/Subscribe Messaging

Before discussing the specifics of Apache Kafka, it is important for us to understand the concept of publish/subscribe messaging and why it is important. *Publish/subscribe messaging* is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/sub systems often have a broker, a central point where messages are published, to facilitate this.

How It Starts

Many use cases for publish/subscribe start out the same way: with a simple message queue or interprocess communication channel. For example, you create an application that needs to send monitoring information somewhere, so you write in a direct connection from your application to an app that displays your metrics on a dashboard, and push metrics over that connection, as seen in [Figure 1-1](#).

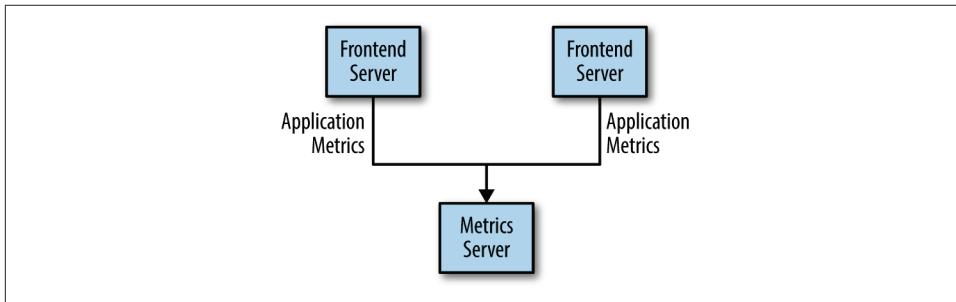


Figure 1-1. A single, direct metrics publisher

This is a simple solution to a simple problem that works when you are getting started with monitoring. Before long, you decide you would like to analyze your metrics over a longer term, and that doesn't work well in the dashboard. You start a new service that can receive metrics, store them, and analyze them. In order to support this, you modify your application to write metrics to both systems. By now you have three more applications that are generating metrics, and they all make the same connections to these two services. Your coworker thinks it would be a good idea to do active polling of the services for alerting as well, so you add a server on each of the applications to provide metrics on request. After a while, you have more applications that are using those servers to get individual metrics and use them for various purposes. This architecture can look much like [Figure 1-2](#), with connections that are even harder to trace.

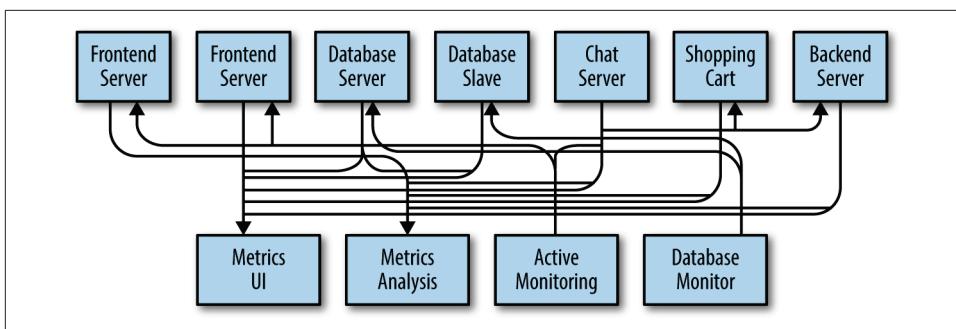


Figure 1-2. Many metrics publishers, using direct connections

The technical debt built up here is obvious, so you decide to pay some of it back. You set up a single application that receives metrics from all the applications out there, and provide a server to query those metrics for any system that needs them. This reduces the complexity of the architecture to something similar to [Figure 1-3](#). Congratulations, you have built a publish-subscribe messaging system!

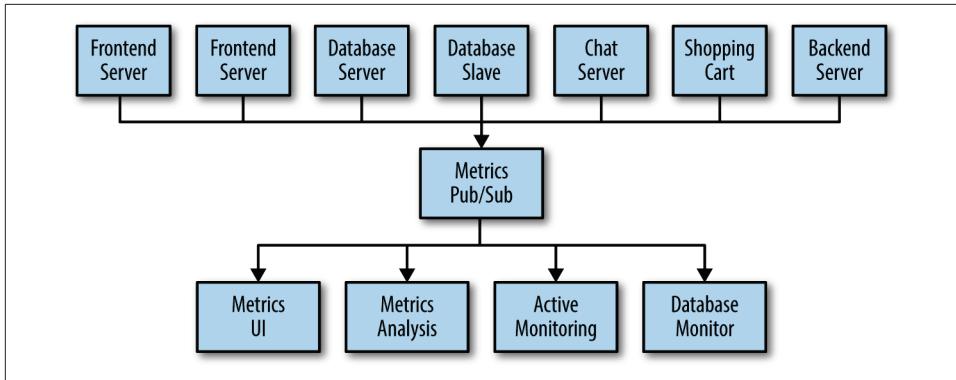


Figure 1-3. A metrics publish/subscribe system

Individual Queue Systems

At the same time that you have been waging this war with metrics, one of your coworkers has been doing similar work with log messages. Another has been working on tracking user behavior on the frontend website and providing that information to developers who are working on machine learning, as well as creating some reports for management. You have all followed a similar path of building out systems that decouple the publishers of the information from the subscribers to that information. [Figure 1-4](#) shows such an infrastructure, with three separate pub/sub systems.

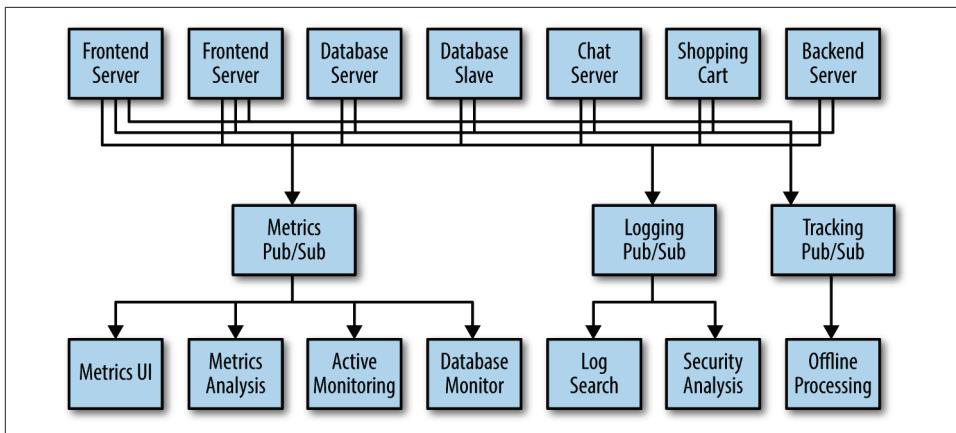


Figure 1-4. Multiple publish/subscribe systems

This is certainly a lot better than utilizing point-to-point connections (as in [Figure 1-2](#)), but there is a lot of duplication. Your company is maintaining multiple systems for queuing data, all of which have their own individual bugs and limitations. You also know that there will be more use cases for messaging coming soon. What you would like to have is a single centralized system that allows for publishing generic types of data, which will grow as your business grows.

Enter Kafka

Apache Kafka is a publish/subscribe messaging system designed to solve this problem. It is often described as a “distributed commit log” or more recently as a “distributing streaming platform.” A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Messages and Batches

The unit of data within Kafka is called a *message*. If you are approaching Kafka from a database background, you can think of this as similar to a *row* or a *record*. A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. A message can have an optional bit of metadata, which is referred to as a *key*. The key is also a byte array and, as with the message, has no specific meaning to Kafka. Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to generate a consistent hash of the key, and then select the partition number for that message by taking the result of the hash modulo, the total number of partitions in the topic. This assures that messages with the same key are always written to the same partition. Keys are discussed in more detail in [Chapter 3](#).

For efficiency, messages are written into Kafka in batches. A *batch* is just a collection of messages, all of which are being produced to the same topic and partition. An individual roundtrip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this. Of course, this is a tradeoff between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, providing more efficient data transfer and storage at the cost of some processing power.

Schemas

While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood. There are many options available for message *schema*, depending on your application's individual needs. Simplistic systems, such as Javascript Object Notation (JSON) and Extensible Markup Language (XML), are easy to use and human-readable. However, they lack features such as robust type handling and compatibility between schema versions. Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop. Avro provides a compact serialization format; schemas that are separate from the message payloads and that do not require code to be generated when they change; and strong data typing and schema evolution, with both backward and forward compatibility.

A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled. When these tasks are tightly coupled, applications that subscribe to messages must be updated to handle the new data format, in parallel with the old format. Only then can the applications that publish the messages be updated to utilize the new format. By using well-defined schemas and storing them in a common repository, the messages in Kafka can be understood without coordination. Schemas and serialization are covered in more detail in [Chapter 3](#).

Topics and Partitions

Messages in Kafka are categorized into *topics*. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of *partitions*. Going back to the “commit log” description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message time-ordering across the entire topic, just within a single partition. [Figure 1-5](#) shows a topic with four partitions, with writes being appended to the end of each one. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.

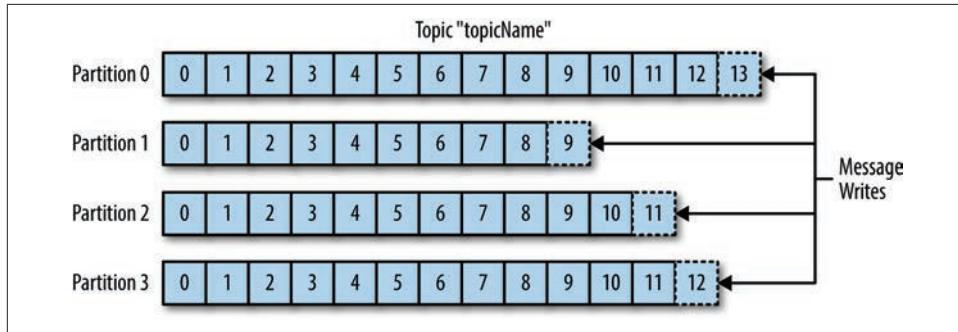


Figure 1-5. Representation of a topic with multiple partitions

The term *stream* is often used when discussing data within systems like Kafka. Most often, a stream is considered to be a single topic of data, regardless of the number of partitions. This represents a single stream of data moving from the producers to the consumers. This way of referring to messages is most common when discussing stream processing, which is when frameworks—some of which are Kafka Streams, Apache Samza, and Storm—operate on the messages in real time. This method of operation can be compared to the way offline frameworks, namely Hadoop, are designed to work on bulk data at a later time. An overview of stream processing is provided in [Chapter 11](#).

Producers and Consumers

Kafka clients are users of the system, and there are two basic types: producers and consumers. There are also advanced client APIs—Kafka Connect API for data integration and Kafka Streams for stream processing. The advanced clients use producers and consumers as building blocks and provide higher-level functionality on top.

Producers create new messages. In other publish/subscribe systems, these may be called *publishers* or *writers*. In general, a message will be produced to a specific topic. By default, the producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions. Producers are covered in more detail in [Chapter 3](#).

Consumers read messages. In other publish/subscribe systems, these clients may be called *subscribers* or *readers*. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the offset of

messages. The *offset* is another bit of metadata—an integer value that continually increases—that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset. By storing the offset of the last consumed message for each partition, either in Zookeeper or in Kafka itself, a consumer can stop and restart without losing its place.

Consumers work as part of a *consumer group*, which is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member. In Figure 1-6, there are three consumers in a single group consuming a topic. Two of the consumers are working from one partition each, while the third consumer is working from two partitions. The mapping of a consumer to a partition is often called *ownership* of the partition by the consumer.

In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in Chapter 4.

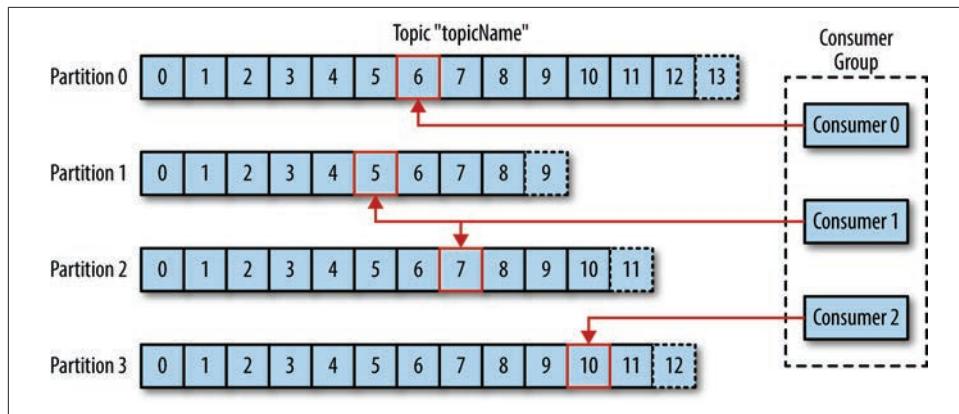


Figure 1-6. A consumer group reading from a topic

Brokers and Clusters

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka brokers are designed to operate as part of a *cluster*. Within a cluster of brokers, one broker will also function as the cluster *controller* (elected automatically from the live members of the cluster). The controller is responsible for administrative opera-

tions, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the *leader* of the partition. A partition may be assigned to multiple brokers, which will result in the partition being replicated (as seen in Figure 1-7). This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure. However, all consumers and producers operating on that partition must connect to the leader. Cluster operations, including partition replication, are covered in detail in [Chapter 6](#).

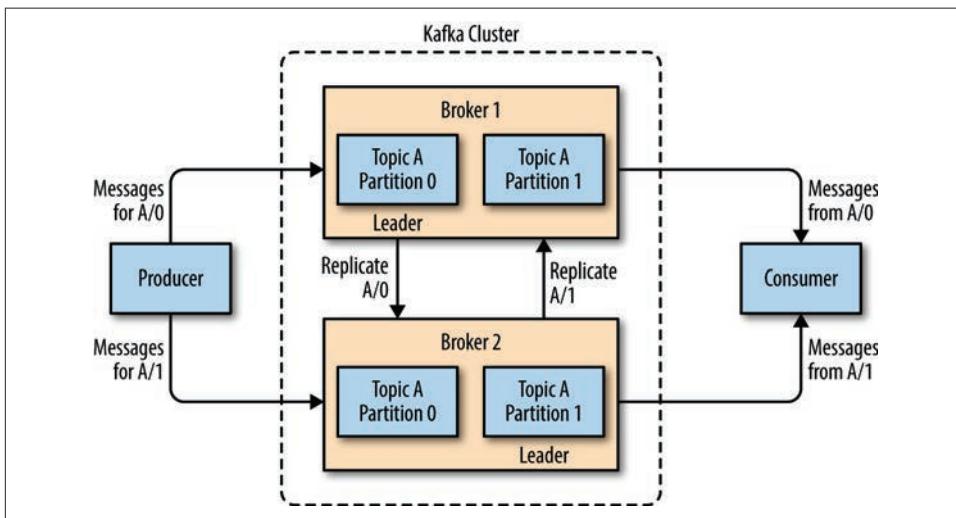


Figure 1-7. Replication of partitions in a cluster

A key feature of Apache Kafka is that of *retention*, which is the durable storage of messages for some period of time. Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the topic reaches a certain size in bytes (e.g., 1 GB). Once these limits are reached, messages are expired and deleted so that the retention configuration is a minimum amount of data available at any time. Individual topics can also be configured with their own retention settings so that messages are stored for only as long as they are useful. For example, a tracking topic might be retained for several days, whereas application metrics might be retained for only a few hours. Topics can also be configured as *log compacted*, which means that Kafka will retain only the last message produced with a specific key. This can be useful for changelog-type data, where only the last update is interesting.

Multiple Clusters

As Kafka deployments grow, it is often advantageous to have multiple clusters. There are several reasons why this can be useful:

- Segregation of types of data
- Isolation for security requirements
- Multiple datacenters (disaster recovery)

When working with multiple datacenters in particular, it is often required that messages be copied between them. In this way, online applications can have access to user activity at both sites. For example, if a user changes public information in their profile, that change will need to be visible regardless of the datacenter in which search results are displayed. Or, monitoring data can be collected from many sites into a single central location where the analysis and alerting systems are hosted. The replication mechanisms within the Kafka clusters are designed only to work within a single cluster, not between multiple clusters.

The Kafka project includes a tool called *MirrorMaker*, used for this purpose. At its core, MirrorMaker is simply a Kafka consumer and producer, linked together with a queue. Messages are consumed from one Kafka cluster and produced for another. [Figure 1-8](#) shows an example of an architecture that uses MirrorMaker, aggregating messages from two local clusters into an aggregate cluster, and then copying that cluster to other datacenters. The simple nature of the application belies its power in creating sophisticated data pipelines, which will be detailed further in [Chapter 7](#).

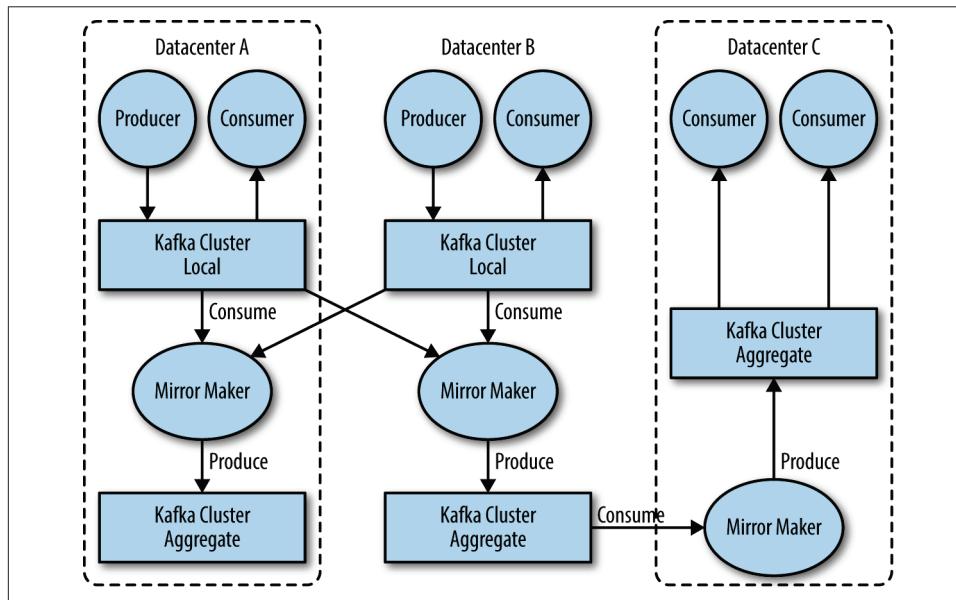


Figure 1-8. Multiple datacenter architecture

Why Kafka?

There are many choices for publish/subscribe messaging systems, so what makes Apache Kafka a good choice?

Multiple Producers

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic. This makes the system ideal for aggregating data from many frontend systems and making it consistent. For example, a site that serves content to users via a number of microservices can have a single topic for page views that all services can write to using a common format. Consumer applications can then receive a single stream of page views for all applications on the site without having to coordinate consuming from multiple topics, one for each application.

Multiple Consumers

In addition to multiple producers, Kafka is designed for multiple consumers to read any single stream of messages without interfering with each other. This is in contrast to many queuing systems where once a message is consumed by one client, it is not available to any other. Multiple Kafka consumers can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Disk-Based Retention

Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time. Messages are committed to disk, and will be stored with configurable retention rules. These options can be selected on a per-topic basis, allowing for different streams of messages to have different amounts of retention depending on the consumer needs. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data. It also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost. Consumers can be stopped, and the messages will be retained in Kafka. This allows them to restart and pick up processing messages where they left off with no data loss.

Scalable

Kafka's flexible scalability makes it easy to handle any amount of data. Users can start with a single broker as a proof of concept, expand to a small development cluster of three brokers, and move into production with a larger cluster of tens or even hundreds of brokers that grows over time as the data scales up. Expansions can be per-

formed while the cluster is online, with no impact on the availability of the system as a whole. This also means that a cluster of multiple brokers can handle the failure of an individual broker, and continue servicing clients. Clusters that need to tolerate more simultaneous failures can be configured with higher replication factors. Replication is discussed in more detail in [Chapter 6](#).

High Performance

All of these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease. This can be done while still providing subsecond message latency from producing a message to availability to consumers.

The Data Ecosystem

Many applications participate in the environments we build for data processing. We have defined inputs in the form of applications that create data or otherwise introduce it to the system. We have defined outputs in the form of metrics, reports, and other data products. We create loops, with some components reading data from the system, transforming it using data from other sources, and then introducing it back into the data infrastructure to be used elsewhere. This is done for numerous types of data, with each having unique qualities of content, size, and usage.

Apache Kafka provides the circulatory system for the data ecosystem, as shown in [Figure 1-9](#). It carries messages between the various members of the infrastructure, providing a consistent interface for all clients. When coupled with a system to provide message schemas, producers and consumers no longer require tight coupling or direct connections of any sort. Components can be added and removed as business cases are created and dissolved, and producers do not need to be concerned about who is using the data or the number of consuming applications.

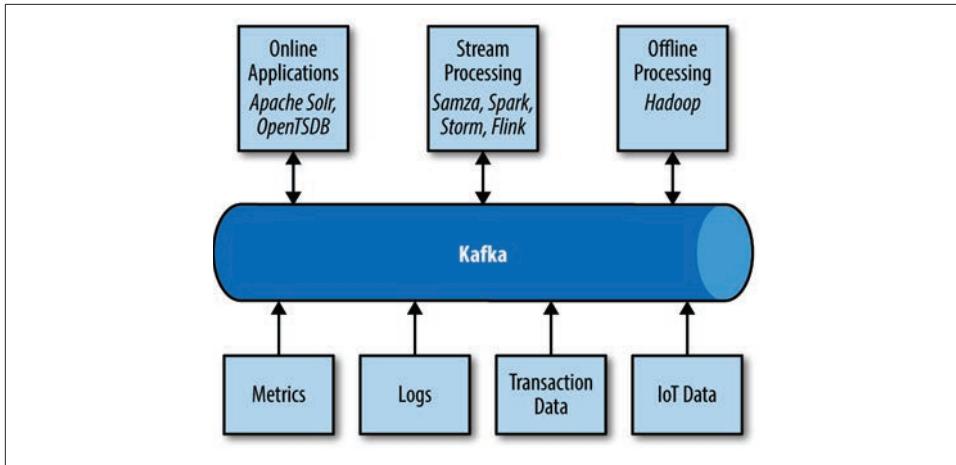


Figure 1-9. A big data ecosystem

Use Cases

Activity tracking

The original use case for Kafka, as it was designed at LinkedIn, is that of user activity tracking. A website's users interact with frontend applications, which generate messages regarding actions the user is taking. This can be passive information, such as page views and click tracking, or it can be more complex actions, such as information that a user adds to their profile. The messages are published to one or more topics, which are then consumed by applications on the backend. These applications may be generating reports, feeding machine learning systems, updating search results, or performing other operations that are necessary to provide a rich user experience.

Messaging

Kafka is also used for messaging, where applications need to send notifications (such as emails) to users. Those applications can produce messages without needing to be concerned about formatting or how the messages will actually be sent. A single application can then read all the messages to be sent and handle them consistently, including:

- Formatting the messages (also known as decorating) using a common look and feel
- Collecting multiple messages into a single notification to be sent
- Applying a user's preferences for how they want to receive messages

Using a single application for this avoids the need to duplicate functionality in multiple applications, as well as allows operations like aggregation which would not otherwise be possible.

Metrics and logging

Kafka is also ideal for collecting application and system metrics and logs. This is a use case in which the ability to have multiple applications producing the same type of message shines. Applications publish metrics on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting. They can also be used in an offline system like Hadoop to perform longer-term analysis, such as growth projections. Log messages can be published in the same way, and can be routed to dedicated log search systems like Elasticsearch or security analysis applications. Another added benefit of Kafka is that when the destination system needs to change (e.g., it's time to update the log storage system), there is no need to alter the frontend applications or the means of aggregation.

Commit log

Since Kafka is based on the concept of a commit log, database changes can be published to Kafka and applications can easily monitor this stream to receive live updates as they happen. This changelog stream can also be used for replicating database updates to a remote system, or for consolidating changes from multiple applications into a single database view. Durable retention is useful here for providing a buffer for the changelog, meaning it can be replayed in the event of a failure of the consuming applications. Alternately, log-compacted topics can be used to provide longer retention by only retaining a single change per key.

Stream processing

Another area that provides numerous types of applications is stream processing. While almost all usage of Kafka can be thought of as stream processing, the term is typically used to refer to applications that provide similar functionality to map/reduce processing in Hadoop. Hadoop usually relies on aggregation of data over a long time frame, either hours or days. Stream processing operates on data in real time, as quickly as messages are produced. Stream frameworks allow users to write small applications to operate on Kafka messages, performing tasks such as counting metrics, partitioning messages for efficient processing by other applications, or transforming messages using data from multiple sources. Stream processing is covered in [Chapter 11](#).

Kafka's Origin

Kafka was created to address the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system that can handle many types of data and provide clean, structured data about user activity and system metrics in real time.

Data really powers everything that we do.

—Jeff Weiner, *CEO of LinkedIn*

LinkedIn's Problem

Similar to the example described at the beginning of this chapter, LinkedIn had a system for collecting system and application metrics that used custom collectors and open source tools for storing and presenting data internally. In addition to traditional metrics, such as CPU usage and application performance, there was a sophisticated request-tracing feature that used the monitoring system and could provide introspection into how a single user request propagated through internal applications. The monitoring system had many faults, however. This included metrics collection based on polling, large intervals between metrics, and no ability for application owners to manage their own metrics. The system was high-touch, requiring human intervention for most simple tasks, and inconsistent, with differing metric names for the same measurement across different systems.

At the same time, there was a system created for tracking user activity information. This was an HTTP service that frontend servers would connect to periodically and publish a batch of messages (in XML format) to the HTTP service. These batches were then moved to offline processing, which is where the files were parsed and collated. This system had many faults. The XML formatting was inconsistent, and parsing it was computationally expensive. Changing the type of user activity that was tracked required a significant amount of coordinated work between frontends and offline processing. Even then, the system would break constantly due to changing schemas. Tracking was built on hourly batching, so it could not be used in real-time.

Monitoring and user-activity tracking could not use the same backend service. The monitoring service was too clunky, the data format was not oriented for activity tracking, and the polling model for monitoring was not compatible with the push model for tracking. At the same time, the tracking service was too fragile to use for metrics, and the batch-oriented processing was not the right model for real-time monitoring and alerting. However, the monitoring and tracking data shared many traits, and correlation of the information (such as how specific types of user activity affected application performance) was highly desirable. A drop in specific types of user activity could indicate problems with the application that serviced it, but hours of delay in processing activity batches meant a slow response to these types of issues.

At first, existing off-the-shelf open source solutions were thoroughly investigated to find a new system that would provide real-time access to the data and scale out to handle the amount of message traffic needed. Prototype systems were set up using ActiveMQ, but at the time it could not handle the scale. It was also a fragile solution for the way LinkedIn needed to use it, discovering many flaws in ActiveMQ that would cause the brokers to pause. This would back up connections to clients and interfere with the ability of the applications to serve requests to users. The decision was made to move forward with a custom infrastructure for the data pipeline.

The Birth of Kafka

The development team at LinkedIn was led by Jay Kreps, a principal software engineer who was previously responsible for the development and open source release of Voldemort, a distributed key-value storage system. The initial team also included Neha Narkhede and, later, Jun Rao. Together, they set out to create a messaging system that could meet the needs of both the monitoring and tracking systems, and scale for the future. The primary goals were to:

- Decouple producers and consumers by using a push-pull model
- Provide persistence for message data within the messaging system to allow multiple consumers
- Optimize for high throughput of messages
- Allow for horizontal scaling of the system to grow as the data streams grew

The result was a publish/subscribe messaging system that had an interface typical of messaging systems but a storage layer more like a log-aggregation system. Combined with the adoption of Apache Avro for message serialization, Kafka was effective for handling both metrics and user-activity tracking at a scale of billions of messages per day. The scalability of Kafka has helped LinkedIn's usage grow in excess of one trillion messages produced (as of August 2015) and over a petabyte of data consumed daily.

Open Source

Kafka was released as an open source project on GitHub in late 2010. As it started to gain attention in the open source community, it was proposed and accepted as an Apache Software Foundation incubator project in July of 2011. Apache Kafka graduated from the incubator in October of 2012. Since then, it has continuously been worked on and has found a robust community of contributors and committers outside of LinkedIn. Kafka is now used in some of the largest data pipelines in the world. In the fall of 2014, Jay Kreps, Neha Narkhede, and Jun Rao left LinkedIn to found Confluent, a company centered around providing development, enterprise support, and training for Apache Kafka. The two companies, along with ever-growing contri-

butions from others in the open source community, continue to develop and maintain Kafka, making it the first choice for big data pipelines.

The Name

People often ask how Kafka got its name and if it has anything to do with the application itself. Jay Kreps offered the following insight:

I thought that since Kafka was a system optimized for writing, using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.

So basically there is not much of a relationship.

Getting Started with Kafka

Now that we know all about Kafka and its history, we can set it up and build our own data pipeline. In the next chapter, we will explore installing and configuring Kafka. We will also cover selecting the right hardware to run Kafka on, and some things to keep in mind when moving to production operations.

CHAPTER 3

Kafka Producers: Writing Messages to Kafka

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by writing a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles.

For example, in a credit card transaction processing system, there will be a client application, perhaps an online store, responsible for sending each transaction to Kafka immediately when a payment is made. Another application is responsible for immediately checking this transaction against a rules engine and determining whether the transaction is approved or denied. The approve/deny response can then be written back to Kafka and the response can propagate back to the online store where the transaction was initiated. A third application can read both transactions and the approval status from Kafka and store them in a database where analysts can later review the decisions and perhaps improve the rules engine.

Apache Kafka ships with built-in client APIs that developers can use when developing applications that interact with Kafka.

In this chapter we will learn how to use the Kafka producer, starting with an overview of its design and components. We will show how to create `KafkaProducer` and `ProducerRecord` objects, how to send records to Kafka, and how to handle the errors that Kafka may return. We'll then review the most important configuration options used to control the producer behavior. We'll conclude with a deeper look at how to use different partitioning methods and serializers, and how to write your own serializers and partitioners.

In [Chapter 4](#) we will look at Kafka's consumer client and reading data from Kafka.



Third-Party Clients

In addition to the built-in clients, Kafka has a binary wire protocol. This means that it is possible for applications to read messages from Kafka or write messages to Kafka simply by sending the correct byte sequences to Kafka's network port. There are multiple clients that implement Kafka's wire protocol in different programming languages, giving simple ways to use Kafka not just in Java applications but also in languages like C++, Python, Go, and many more. Those clients are not part of Apache Kafka project, but a list of non-Java clients is maintained in the [project wiki](#). The wire protocol and the external clients are outside the scope of the chapter.

Producer Overview

There are many reasons an application might need to write messages to Kafka: recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, communicating asynchronously with other applications, buffering information before writing to a database, and much more.

Those diverse use cases also imply diverse requirements: is every message critical, or can we tolerate loss of messages? Are we OK with accidentally duplicating messages? Are there any strict latency or throughput requirements we need to support?

In the credit card transaction processing example we introduced earlier, we can see that it is critical to never lose a single message nor duplicate any messages. Latency should be low but latencies up to 500ms can be tolerated, and throughput should be very high—we expect to process up to a million messages a second.

A different use case might be to store click information from a website. In that case, some message loss or a few duplicates can be tolerated; latency can be high as long as there is no impact on the user experience. In other words, we don't mind if it takes a few seconds for the message to arrive at Kafka, as long as the next page loads immediately after the user clicked on a link. Throughput will depend on the level of activity we anticipate on our website.

The different requirements will influence the way you use the producer API to write messages to Kafka and the configuration you use.

While the producer APIs are very simple, there is a bit more that goes on under the hood of the producer when we send data. [Figure 3-1](#) shows the main steps involved in sending data to Kafka.

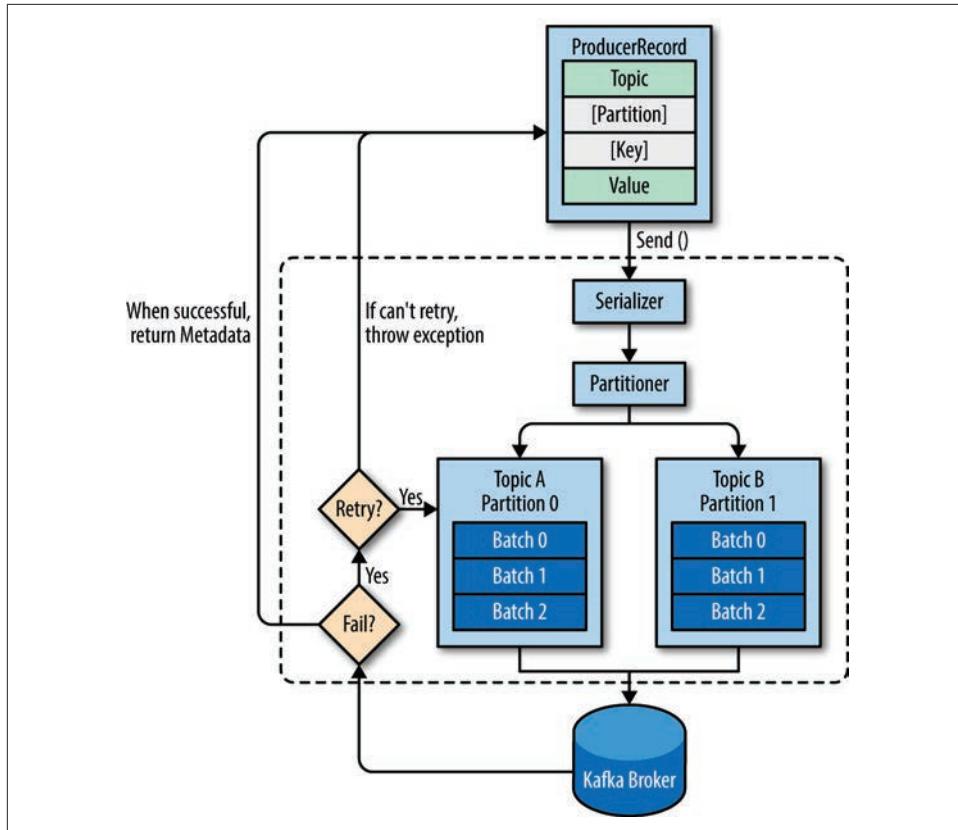


Figure 3-1. High-level overview of Kafka producer components

We start producing messages to Kafka by creating a `ProducerRecord`, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key and/or a partition. Once we send the `ProducerRecord`, the first thing the producer will do is serialize the key and value objects to `ByteArrays` so they can be sent over the network.

Next, the data is sent to a partitioner. If we specified a partition in the `ProducerRecord`, the partitioner doesn't do anything and simply returns the partition we specified. If we didn't, the partitioner will choose a partition for us, usually based on the `ProducerRecord` key. Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition. A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a `RecordMetadata` object with the

topic, partition, and the offset of the record within the partition. If the broker failed to write the messages, it will return an error. When the producer receives an error, it may retry sending the message a few more times before giving up and returning an error.

Constructing a Kafka Producer

The first step in writing messages to Kafka is to create a producer object with the properties you want to pass to the producer. A Kafka producer has three mandatory properties:

`bootstrap.servers`

List of host:port pairs of brokers that the producer will use to establish initial connection to the Kafka cluster. This list doesn't need to include all brokers, since the producer will get more information after the initial connection. But it is recommended to include at least two, so in case one broker goes down, the producer will still be able to connect to the cluster.

`key.serializer`

Name of a class that will be used to serialize the keys of the records we will produce to Kafka. Kafka brokers expect byte arrays as keys and values of messages. However, the producer interface allows, using parameterized types, any Java object to be sent as a key and value. This makes for very readable code, but it also means that the producer has to know how to convert these objects to byte arrays. `key.serializer` should be set to a name of a class that implements the `org.apache.kafka.common.serialization.Serializer` interface. The producer will use this class to serialize the key object to a byte array. The Kafka client package includes `ByteArraySerializer` (which doesn't do much), `StringSerializer`, and `IntegerSerializer`, so if you use common types, there is no need to implement your own serializers. Setting `key.serializer` is required even if you intend to send only values.

`value.serializer`

Name of a class that will be used to serialize the values of the records we will produce to Kafka. The same way you set `key.serializer` to a name of a class that will serialize the message key object to a byte array, you set `value.serializer` to a class that will serialize the message value object.

The following code snippet shows how to create a new producer by setting just the mandatory parameters and using defaults for everything else:

```
private Properties kafkaProps = new Properties(); ①
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
```

```

    "org.apache.kafka.common.serialization.StringSerializer"); ②
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps); ③

```

- ① We start with a `Properties` object.
- ② Since we plan on using strings for message key and value, we use the built-in `StringSerializer`.
- ③ Here we create a new producer by setting the appropriate key and value types and passing the `Properties` object.

With such a simple interface, it is clear that most of the control over producer behavior is done by setting the correct configuration properties. Apache Kafka documentation covers all the [configuration options](#), and we will go over the important ones later in this chapter.

Once we instantiate a producer, it is time to start sending messages. There are three primary methods of sending messages:

Fire-and-forget

We send a message to the server and don't really care if it arrives successfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, some messages will get lost using this method.

Synchronous send

We send a message, the `send()` method returns a `Future` object, and we use `get()` to wait on the future and see if the `send()` was successful or not.

Asynchronous send

We call the `send()` method with a callback function, which gets triggered when it receives a response from the Kafka broker.

In the examples that follow, we will see how to send messages using these methods and how to handle the different types of errors that might occur.

While all the examples in this chapter are single threaded, a producer object can be used by multiple threads to send messages. You will probably want to start with one producer and one thread. If you need better throughput, you can add more threads that use the same producer. Once this ceases to increase throughput, you can add more producers to the application to achieve even higher throughput.

Sending a Message to Kafka

The simplest way to send a message is as follows:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products",
    "France"); ①
try {
    producer.send(record); ②
} catch (Exception e) {
    e.printStackTrace(); ③
}
```

- ❶ The producer accepts `ProducerRecord` objects, so we start by creating one. `ProducerRecord` has multiple constructors, which we will discuss later. Here we use one that requires the name of the topic we are sending data to, which is always a string, and the key and value we are sending to Kafka, which in this case are also strings. The types of the key and value must match our `serializer` and `producer` objects.
- ❷ We use the producer object `send()` method to send the `ProducerRecord`. As we've seen in the producer architecture diagram in [Figure 3-1](#), the message will be placed in a buffer and will be sent to the broker in a separate thread. The `send()` method returns a [Java Future object](#) with `RecordMetadata`, but since we simply ignore the returned value, we have no way of knowing whether the message was sent successfully or not. This method of sending messages can be used when dropping a message silently is acceptable. This is not typically the case in production applications.
- ❸ While we ignore errors that may occur while sending messages to Kafka brokers or in the brokers themselves, we may still get an exception if the producer encountered errors before sending the message to Kafka. Those can be a `SerializationException` when it fails to serialize the message, a `BufferExhaustedException` or `TimeoutException` if the buffer is full, or an `InterruptedException` if the sending thread was interrupted.

Sending a Message Synchronously

The simplest way to send a message synchronously is as follows:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
try {
    producer.send(record).get(); ①
} catch (Exception e) {
```

```
        e.printStackTrace(); ②  
    }
```

- ❶ Here, we are using `Future.get()` to wait for a reply from Kafka. This method will throw an exception if the record is not sent successfully to Kafka. If there were no errors, we will get a `RecordMetadata` object that we can use to retrieve the offset the message was written to.
- ❷ If there were any errors before sending data to Kafka, while sending, if the Kafka brokers returned a nonretryable exceptions or if we exhausted the available retries, we will encounter an exception. In this case, we just print any exception we ran into.

`KafkaProducer` has two types of errors. *Retriable* errors are those that can be resolved by sending the message again. For example, a connection error can be resolved because the connection may get reestablished. A “no leader” error can be resolved when a new leader is elected for the partition. `KafkaProducer` can be configured to retry those errors automatically, so the application code will get retriable exceptions only when the number of retries was exhausted and the error was not resolved. Some errors will not be resolved by retrying. For example, “message size too large.” In those cases, `KafkaProducer` will not attempt a retry and will return the exception immediately.

Sending a Message Asynchronously

Suppose the network roundtrip time between our application and the Kafka cluster is 10ms. If we wait for a reply after sending each message, sending 100 messages will take around 1 second. On the other hand, if we just send all our messages and not wait for any replies, then sending 100 messages will barely take any time at all. In most cases, we really don’t need a reply—Kafka sends back the topic, partition, and offset of the record after it was written, which is usually not required by the sending app. On the other hand, we do need to know when we failed to send a message completely so we can throw an exception, log an error, or perhaps write the message to an “errors” file for later analysis.

In order to send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record. Here is an example of how we use a callback:

```

private class DemoProducerCallback implements Callback { ❶
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace(); ❷
        }
    }
}

ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");
❸
producer.send(record, new DemoProducerCallback()); ❹

```

- ❶ To use callbacks, you need a class that implements the `org.apache.kafka.clients.producer.Callback` interface, which has a single function—`onCompletion()`.
- ❷ If Kafka returned an error, `onCompletion()` will have a nonnull exception. Here we “handle” it by printing, but production code will probably have more robust error handling functions.
- ❸ The records are the same as before.
- ❹ And we pass a `Callback` object along when sending the record.

Configuring Producers

So far we’ve seen very few configuration parameters for the producers—just the mandatory `bootstrap.servers` URI and serializers.

The producer has a large number of configuration parameters; most are documented in Apache Kafka [documentation](#) and many have reasonable defaults so there is no reason to tinker with every single parameter. However, some of the parameters have a significant impact on memory use, performance, and reliability of the producers. We will review those here.

acks

The `acks` parameter controls how many partition replicas must receive the record before the producer can consider the write successful. This option has a significant impact on how likely messages are to be lost. There are three allowed values for the `acks` parameter:

- If `acks=0`, the producer will not wait for a reply from the broker before assuming the message was sent successfully. This means that if something went wrong and

the broker did not receive the message, the producer will not know about it and the message will be lost. However, because the producer is not waiting for any response from the server, it can send messages as fast as the network will support, so this setting can be used to achieve very high throughput.

- If `acks=1`, the producer will receive a success response from the broker the moment the leader replica received the message. If the message can't be written to the leader (e.g., if the leader crashed and a new leader was not elected yet), the producer will receive an error response and can retry sending the message, avoiding potential loss of data. The message can still get lost if the leader crashes and a replica without this message gets elected as the new leader (via unclean leader election). In this case, throughput depends on whether we send messages synchronously or asynchronously. If our client code waits for a reply from the server (by calling the `get()` method of the `Future` object returned when sending a message) it will obviously increase latency significantly (at least by a network roundtrip). If the client uses callbacks, latency will be hidden, but throughput will be limited by the number of in-flight messages (i.e., how many messages the producer will send before receiving replies from the server).
- If `acks=all`, the producer will receive a success response from the broker once all in-sync replicas received the message. This is the safest mode since you can make sure more than one broker has the message and that the message will survive even in the case of crash (more information on this in [Chapter 5](#)). However, the latency we discussed in the `acks=1` case will be even higher, since we will be waiting for more than just one broker to receive the message.

buffer.memory

This sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers. If messages are sent by the application faster than they can be delivered to the server, the producer may run out of space and additional `send()` calls will either block or throw an exception, based on the `block.on.buffer.full` parameter (replaced with `max.block.ms` in release 0.9.0.0, which allows blocking for a certain time and then throwing an exception).

compression.type

By default, messages are sent uncompressed. This parameter can be set to `snappy`, `gzip`, or `lz4`, in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers. Snappy compression was invented by Google to provide decent compression ratios with low CPU overhead and good performance, so it is recommended in cases where both performance and bandwidth are a concern. Gzip compression will typically use more CPU and time but result in better compression ratios, so it is recommended in cases where network bandwidth is

more restricted. By enabling compression, you reduce network utilization and storage, which is often a bottleneck when sending messages to Kafka.

retries

When the producer receives an error message from the server, the error could be transient (e.g., a lack of leader for a partition). In this case, the value of the `retries` parameter will control how many times the producer will retry sending the message before giving up and notifying the client of an issue. By default, the producer will wait 100ms between retries, but you can control this using the `retry.backoff.ms` parameter. We recommend testing how long it takes to recover from a crashed broker (i.e., how long until all partitions get new leaders) and setting the number of retries and delay between them such that the total amount of time spent retrying will be longer than the time it takes the Kafka cluster to recover from the crash—otherwise, the producer will give up too soon. Not all errors will be retried by the producer. Some errors are not transient and will not cause retries (e.g., “message too large” error). In general, because the producer handles retries for you, there is no point in handling retries within your own application logic. You will want to focus your efforts on handling nonretryable errors or cases where retry attempts were exhausted.

batch.size

When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes (not messages!) that will be used for each batch. When the batch is full, all the messages in the batch will be sent. However, this does not mean that the producer will wait for the batch to become full. The producer will send half-full batches and even batches with just a single message in them. Therefore, setting the batch size too large will not cause delays in sending messages; it will just use more memory for the batches. Setting the batch size too small will add some overhead because the producer will need to send messages more frequently.

linger.ms

`linger.ms` controls the amount of time to wait for additional messages before sending the current batch. `KafkaProducer` sends a batch of messages either when the current batch is full or when the `linger.ms` limit is reached. By default, the producer will send messages as soon as there is a sender thread available to send them, even if there’s just one message in the batch. By setting `linger.ms` higher than 0, we instruct the producer to wait a few milliseconds to add additional messages to the batch before sending it to the brokers. This increases latency but also increases throughput (because we send more messages at once, there is less overhead per message).

client.id

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics, and for quotas.

max.in.flight.requests.per.connection

This controls how many messages the producer will send to the server without receiving responses. Setting this high can increase memory usage while improving throughput, but setting it too high can reduce throughput as batching becomes less efficient. Setting this to 1 will guarantee that messages will be written to the broker in the order in which they were sent, even when retries occur.

timeout.ms, request.timeout.ms, and metadata.fetch.timeout.ms

These parameters control how long the producer will wait for a reply from the server when sending data (`request.timeout.ms`) and when requesting metadata such as the current leaders for the partitions we are writing to (`metadata.fetch.timeout.ms`). If the timeout is reached without reply, the producer will either retry sending or respond with an error (either through exception or the send callback). `timeout.ms` controls the time the broker will wait for in-sync replicas to acknowledge the message in order to meet the `acks` configuration—the broker will return an error if the time elapses without the necessary acknowledgments.

max.block.ms

This parameter controls how long the producer will block when calling `send()` and when explicitly requesting metadata via `partitionsFor()`. Those methods block when the producer's send buffer is full or when metadata is not available. When `max.block.ms` is reached, a timeout exception is thrown.

max.request.size

This setting controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request. For example, with a default maximum request size of 1 MB, the largest message you can send is 1 MB or the producer can batch 1,000 messages of size 1 K each into one request. In addition, the broker has its own limit on the size of the largest message it will accept (`message.max.bytes`). It is usually a good idea to have these configurations match, so the producer will not attempt to send messages of a size that will be rejected by the broker.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the OS defaults will be used. It is a

good idea to increase those when producers or consumers communicate with brokers in a different datacenter because those network links typically have higher latency and lower bandwidth.



Ordering Guarantees

Apache Kafka preserves the order of messages within a partition. This means that if messages were sent from the producer in a specific order, the broker will write them to a partition in that order and all consumers will read them in that order. For some use cases, order is very important. There is a big difference between depositing \$100 in an account and later withdrawing it, and the other way around! However, some use cases are less sensitive.

Setting the `retries` parameter to nonzero and the `max.in.flights.requests.per.session` to more than one means that it is possible that the broker will fail to write the first batch of messages, succeed to write the second (which was already in-flight), and then retry the first batch and succeed, thereby reversing the order.

Usually, setting the number of retries to zero is not an option in a reliable system, so if guaranteeing order is critical, we recommend setting `in.flight.requests.per.session=1` to make sure that while a batch of messages is retrying, additional messages will not be sent (because this has the potential to reverse the correct order). This will severely limit the throughput of the producer, so only use this when order is important.

Serializers

As seen in previous examples, producer configuration includes mandatory serializers. We've seen how to use the default String serializer. Kafka also includes serializers for integers and `ByteArrays`, but this does not cover most use cases. Eventually, you will want to be able to serialize more generic records.

We will start by showing how to write your own serializer and then introduce the Avro serializer as a recommended alternative.

Custom Serializers

When the object you need to send to Kafka is not a simple string or integer, you have a choice of either using a generic serialization library like Avro, Thrift, or Protobuf to create records, or creating a custom serialization for objects you are already using. We highly recommend using a generic serialization library. In order to understand how

the serializers work and why it is a good idea to use a serialization library, let's see what it takes to write your own custom serializer.

Suppose that instead of recording just the customer name, you create a simple class to represent customers:

```
public class Customer {  
    private int customerID;  
    private String customerName;  
  
    public Customer(int ID, String name) {  
        this.customerID = ID;  
        this.customerName = name;  
    }  
  
    public int getID() {  
        return customerID;  
    }  
  
    public String getName() {  
        return customerName;  
    }  
}
```

Now suppose we want to create a custom serializer for this class. It will look something like this:

```
import org.apache.kafka.common.errors.SerializationException;  
  
import java.nio.ByteBuffer;  
import java.util.Map;  
  
public class CustomerSerializer implements Serializer<Customer> {  
  
    @Override  
    public void configure(Map configs, boolean isKey) {  
        // nothing to configure  
    }  
  
    @Override  
    /**  
     * We are serializing Customer as:  
     * 4 byte int representing customerId  
     * 4 byte int representing length of customerName in UTF-8 bytes (0 if name is  
     * Null)  
     * N bytes representing customerName in UTF-8  
     */  
    public byte[] serialize(String topic, Customer data) {  
        try {  
            byte[] serializedName;  
            int stringSize;  
            if (data == null)  
                return null;
```

```

        else {
            if (data.getName() != null) {
                serializedName = data.getName().getBytes("UTF-8");
                stringSize = serializedName.length;
            } else {
                serializedName = new byte[0];
                stringSize = 0;
            }
        }

        ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
        buffer.putInt(data.getID());
        buffer.putInt(stringSize);
        buffer.put(serializedName);

        return buffer.array();
    } catch (Exception e) {
        throw new SerializationException("Error when serializing Customer to
byte[] " + e);
    }
}

@Override
public void close() {
    // nothing to close
}
}

```

Configuring a producer with this `CustomerSerializer` will allow you to define `ProducerRecord<String, Customer>`, and send `Customer` data and pass `Customer` objects directly to the producer. This example is pretty simple, but you can see how fragile the code is. If we ever have too many customers, for example, and need to change `customerID` to `Long`, or if we ever decide to add a `startDate` field to `Customer`, we will have a serious issue in maintaining compatibility between old and new messages. Debugging compatibility issues between different versions of serializers and deserializers is fairly challenging—you need to compare arrays of raw bytes. To make matters even worse, if multiple teams in the same company end up writing `Customer` data to Kafka, they will all need to use the same serializers and modify the code at the exact same time.

For these reasons, we recommend using existing serializers and deserializers such as JSON, Apache Avro, Thrift, or Protobuf. In the following section we will describe Apache Avro and then show how to serialize Avro records and send them to Kafka.

Serializing Using Apache Avro

Apache Avro is a language-neutral data serialization format. The project was created by Doug Cutting to provide a way to share data files with a large audience.

Avro data is described in a language-independent schema. The schema is usually described in JSON and the serialization is usually to binary files, although serializing to JSON is also supported. Avro assumes that the schema is present when reading and writing files, usually by embedding the schema in the files themselves.

One of the most interesting features of Avro, and what makes it a good fit for use in a messaging system like Kafka, is that when the application that is writing messages switches to a new schema, the applications reading the data can continue processing messages without requiring any change or update.

Suppose the original schema was:

```
{"namespace": "customerManagement.avro",
"type": "record",
"name": "Customer",
"fields": [
    {"name": "id", "type": "int"},
    {"name": "name", "type": "string"},
    {"name": "faxNumber", "type": ["null", "string"], "default": "null"} ❶
]
}
```

- ❶ id and name fields are mandatory, while fax number is optional and defaults to null.

We used this schema for a few months and generated a few terabytes of data in this format. Now suppose that we decide that in the new version, we will upgrade to the twenty-first century and will no longer include a fax number field and will instead use an email field.

The new schema would be:

```
{"namespace": "customerManagement.avro",
"type": "record",
"name": "Customer",
"fields": [
    {"name": "id", "type": "int"},
    {"name": "name", "type": "string"},
    {"name": "email", "type": ["null", "string"], "default": "null"}
]
}
```

Now, after upgrading to the new version, old records will contain “faxNumber” and new records will contain “email.” In many organizations, upgrades are done slowly and over many months. So we need to consider how preupgrade applications that still use the fax numbers and postupgrade applications that use email will be able to handle all the events in Kafka.

The reading application will contain calls to methods similar to `getName()`, `getId()`, and `getFaxNumber`. If it encounters a message written with the new schema, get

`Name()` and `getId()` will continue working with no modification, but `getFaxNumber()` will return `null` because the message will not contain a fax number.

Now suppose we upgrade our reading application and it no longer has the `getFaxNumber()` method but rather `getEmail()`. If it encounters a message written with the old schema, `getEmail()` will return `null` because the older messages do not contain an email address.

This example illustrates the benefit of using Avro: even though we changed the schema in the messages without changing all the applications reading the data, there will be no exceptions or breaking errors and no need for expensive updates of existing data.

However, there are two caveats to this scenario:

- The schema used for writing the data and the schema expected by the reading application must be compatible. The Avro documentation includes [compatibility rules](#).
- The deserializer will need access to the schema that was used when writing the data, even when it is different than the schema expected by the application that accesses the data. In Avro files, the writing schema is included in the file itself, but there is a better way to handle this for Kafka messages. We will look at that next.

Using Avro Records with Kafka

Unlike Avro files, where storing the entire schema in the data file is associated with a fairly reasonable overhead, storing the entire schema in each record will usually more than double the record size. However, Avro still requires the entire schema to be present when reading the record, so we need to locate the schema elsewhere. To achieve this, we follow a common architecture pattern and use a *Schema Registry*. The Schema Registry is not part of Apache Kafka but there are several open source options to choose from. We'll use the Confluent Schema Registry for this example. You can find the Schema Registry code on [GitHub](#), or you can install it as part of the [Confluent Platform](#). If you decide to use the Schema Registry, then we recommend checking the [documentation](#).

The idea is to store all the schemas used to write data to Kafka in the registry. Then we simply store the identifier for the schema in the record we produce to Kafka. The consumers can then use the identifier to pull the record out of the schema registry and deserialize the data. The key is that all this work—storing the schema in the registry and pulling it up when required—is done in the serializers and deserializers. The code that produces data to Kafka simply uses the Avro serializer just like it would any other serializer. [Figure 3-2](#) demonstrates this process.

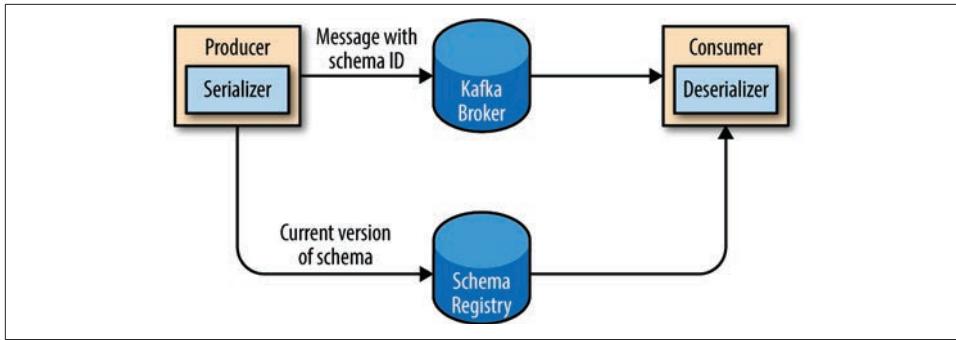


Figure 3-2. Flow diagram of serialization and deserialization of Avro records

Here is an example of how to produce generated Avro objects to Kafka (see the [Avro Documentation](#) for how to use code generation with Avro):

```

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
        "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer",
        "io.confluent.kafka.serializers.KafkaAvroSerializer"); ①
props.put("schema.registry.url", schemaUrl); ②

String topic = "customerContacts";
int wait = 500;

Producer<String, Customer> producer = new KafkaProducer<String,
Customer>(props); ③

// We keep producing new events until someone ctrl-c
while (true) {
    Customer customer = CustomerGenerator.getNext();
    System.out.println("Generated customer " +
    customer.toString());
    ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getId(), cus-
    tomer); ④
    producer.send(record); ⑤
}

```

- ① We use the `KafkaAvroSerializer` to serialize our objects with Avro. Note that the `AvroSerializer` can also handle primitives, which is why we can later use `String` as the record key and our `Customer` object as the value.
- ② `schema.registry.url` is a new parameter. This simply points to where we store the schemas.

- ③ `Customer` is our generated object. We tell the producer that our records will contain `Customer` as the value.
- ④ We also instantiate `ProducerRecord` with `Customer` as the value type, and pass a `Customer` object when creating the new record.
- ⑤ That's it. We send the record with our `Customer` object and `KafkaAvroSerializer` will handle the rest.

What if you prefer to use generic Avro objects rather than the generated Avro objects? No worries. In this case, you just need to provide the schema:

```

Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
          "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("value.serializer",
          "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

String schemaString = "{\"namespace\": \"customerManagement.avro\",
                      \"type\": \"record\", " + ❸
                      "\"name\": \"Customer\", " +
                      "\"fields\": [ " +
                      "{\"name\": \"id\", \"type\": \"int\"}, " +
                      "{\"name\": \"name\", \"type\": \"string\"}, " +
                      "{\"name\": \"email\", \"type\": [\"null\", \"string
                      \"]}, \"default\": \"null\" }" +
                      "]}";
Producer<String, GenericRecord> producer =
    new KafkaProducer<String, GenericRecord>(props); ❹

Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(schemaString);

for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
    String name = "exampleCustomer" + nCustomers;
    String email = "example " + nCustomers + "@example.com"

    GenericRecord customer = new GenericData.Record(schema); ❺
    customer.put("id", nCustomer);
    customer.put("name", name);
    customer.put("email", email);

    ProducerRecord<String, GenericRecord> data =
        new ProducerRecord<String,
                           GenericRecord>("customerContacts",
                           name, customer);
    producer.send(data);
}

```

```
    }  
}
```

- ➊ We still use the same `KafkaAvroSerializer`.
- ➋ And we provide the URI of the same schema registry.
- ➌ But now we also need to provide the Avro schema, since it is not provided by the Avro-generated object.
- ➍ Our object type is an Avro `GenericRecord`, which we initialize with our schema and the data we want to write.
- ➎ Then the value of the `ProducerRecord` is simply a `GenericRecord` that contains our schema and data. The serializer will know how to get the schema from this record, store it in the schema registry, and serialize the object data.

Partitions

In previous examples, the `ProducerRecord` objects we created included a topic name, key, and value. Kafka messages are key-value pairs and while it is possible to create a `ProducerRecord` with just a topic and a value, with the key set to `null` by default, most applications produce records with keys. Keys serve two goals: they are additional information that gets stored with the message, and they are also used to decide which one of the topic partitions the message will be written to. All messages with the same key will go to the same partition. This means that if a process is reading only a subset of the partitions in a topic (more on that in [Chapter 4](#)), all the records for a single key will be read by the same process. To create a key-value record, you simply create a `ProducerRecord` as follows:

```
ProducerRecord<Integer, String> record =  
    new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "USA");
```

When creating messages with a `null` key, you can simply leave the key out:

```
ProducerRecord<Integer, String> record =  
    new ProducerRecord<>("CustomerCountry", "USA"); ➊
```

- ➊ Here, the key will simply be set to `null`, which may indicate that a customer name was missing on a form.

When the key is `null` and the default partitioner is used, the record will be sent to one of the available partitions of the topic at random. A round-robin algorithm will be used to balance the messages among the partitions.

If a key exists and the default partitioner is used, Kafka will hash the key (using its own hash algorithm, so hash values will not change when Java is upgraded), and use the result to map the message to a specific partition. Since it is important that a key is always mapped to the same partition, we use all the partitions in the topic to calculate the mapping—not just the available partitions. This means that if a specific partition is unavailable when you write data to it, you might get an error. This is fairly rare, as you will see in [Chapter 6](#) when we discuss Kafka’s replication and availability.

The mapping of keys to partitions is consistent only as long as the number of partitions in a topic does not change. So as long as the number of partitions is constant, you can be sure that, for example, records regarding user 045189 will always get written to partition 34. This allows all kinds of optimization when reading data from partitions. However, the moment you add new partitions to the topic, this is no longer guaranteed—the old records will stay in partition 34 while new records will get written to a different partition. When partitioning keys is important, the easiest solution is to create topics with sufficient partitions ([Chapter 2](#) includes suggestions for how to determine a good number of partitions) and never add partitions.

Implementing a custom partitioning strategy

So far, we have discussed the traits of the default partitioner, which is the one most commonly used. However, Kafka does not limit you to just hash partitions, and sometimes there are good reasons to partition data differently. For example, suppose that you are a B2B vendor and your biggest customer is a company that manufactures handheld devices called Bananas. Suppose that you do so much business with customer “Banana” that over 10% of your daily transactions are with this customer. If you use default hash partitioning, the Banana records will get allocated to the same partition as other accounts, resulting in one partition being about twice as large as the rest. This can cause servers to run out of space, processing to slow down, etc. What we really want is to give Banana its own partition and then use hash partitioning to map the rest of the accounts to partitions.

Here is an example of a custom partitioner:

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

    public void configure(Map<String, ?> configs) {} ①

    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes,
                         Cluster cluster) {
        List<PartitionInfo> partitions =
            cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceof String))) ②
            throw new InvalidRecordException("We expect all messages
                to have customer name as key")

        if (((String) key).equals("Banana"))
            return numPartitions; // Banana will always go to last
                                  partition

        // Other records will get hashed to the rest of the
        // partitions
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
    }

    public void close() {}
}
```

- ① Partitioner interface includes `configure`, `partition`, and `close` methods. Here we only implement `partition`, although we really should have passed the special customer name through `configure` instead of hard-coding it in `partition`.
- ② We only expect String keys, so we throw an exception if that is not the case.

Old Producer APIs

In this chapter we've discussed the Java producer client that is part of the `org.apache.kafka.clients` package. However, Apache Kafka still has two older clients written in Scala that are part of the `kafka.producer` package and the core Kafka module. These producers are called `SyncProducers` (which, depending on the value of the `acks` parameter, may wait for the server to ack each message or batch of messages before sending additional messages) and `AsyncProducer` (which batches mes-

sages in the background, sends them in a separate thread, and does not provide feedback regarding success to the client).

Because the current producer supports both behaviors and provides much more reliability and control to the developer, we will not discuss the older APIs. If you are interested in using them, think twice and then refer to Apache Kafka documentation to learn more.

Summary

We began this chapter with a simple example of a producer—just 10 lines of code that send events to Kafka. We added to the simple example by adding error handling and experimenting with synchronous and asynchronous producing. We then explored the most important producer configuration parameters and saw how they modify the behavior of the producers. We discussed serializers, which let us control the format of the events we write to Kafka. We looked in-depth at Avro, one of many ways to serialize events, but one that is very commonly used with Kafka. We concluded the chapter with a discussion of partitioning in Kafka and an example of an advanced custom partitioning technique.

Now that we know how to write events to Kafka, in [Chapter 4](#) we'll learn all about consuming events from Kafka.

CHAPTER 4

Kafka Consumers: Reading Data from Kafka

Applications that need to read data from Kafka use a `KafkaConsumer` to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems, and there are few unique concepts and ideas involved. It is difficult to understand how to use the consumer API without understanding these concepts first. We'll start by explaining some of the important concepts, and then we'll go through some examples that show the different ways consumer APIs can be used to implement applications with varying requirements.

Kafka Consumer Concepts

In order to understand how to read data from Kafka, you first need to understand its consumers and consumer groups. The following sections cover those concepts.

Consumers and Consumer Groups

Suppose you have an application that needs to read messages from a Kafka topic, run some validations against them, and write the results to another data store. In this case your application will create a consumer object, subscribe to the appropriate topic, and start receiving messages, validating them and writing the results. This may work well for a while, but what if the rate at which producers write messages to the topic exceeds the rate at which your application can validate them? If you are limited to a single consumer reading and processing the data, your application may fall farther and farther behind, unable to keep up with the rate of incoming messages. Obviously there is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data between them.

Kafka consumers are typically part of a consumer group. When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

Let's take topic T1 with four partitions. Now suppose we created a new consumer, C1, which is the only consumer in group G1, and use it to subscribe to topic T1. Consumer C1 will get all messages from all four t1 partitions. See [Figure 4-1](#).

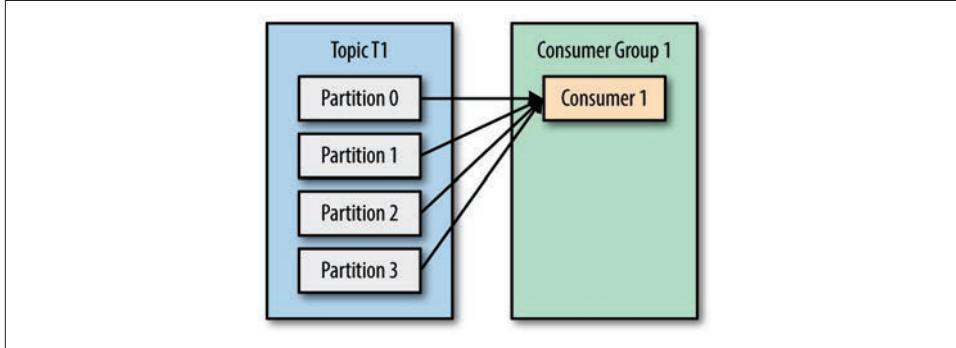


Figure 4-1. One Consumer group with four partitions

If we add another consumer, C2, to group G1, each consumer will only get messages from two partitions. Perhaps messages from partition 0 and 2 go to C1 and messages from partitions 1 and 3 go to consumer C2. See [Figure 4-2](#).

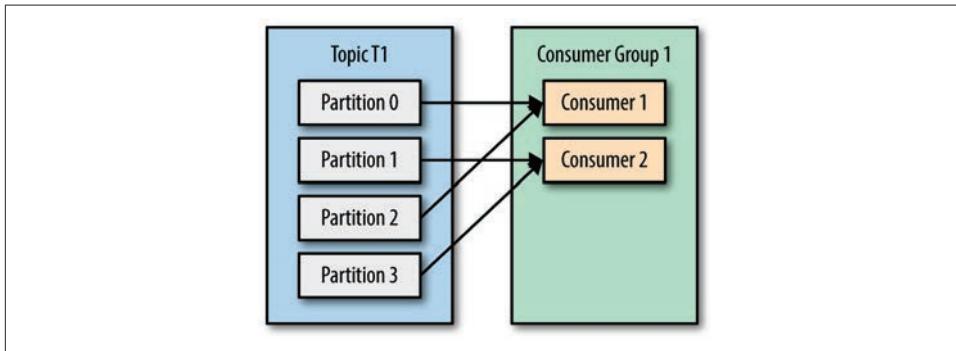


Figure 4-2. Four partitions split to two consumer groups

If G1 has four consumers, then each will read messages from a single partition. See [Figure 4-3](#).

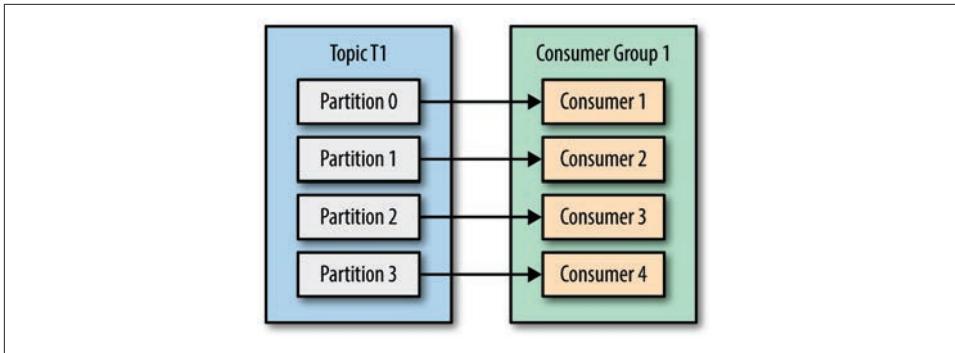


Figure 4-3. Four consumer groups to one partition each

If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all. See [Figure 4-4](#).

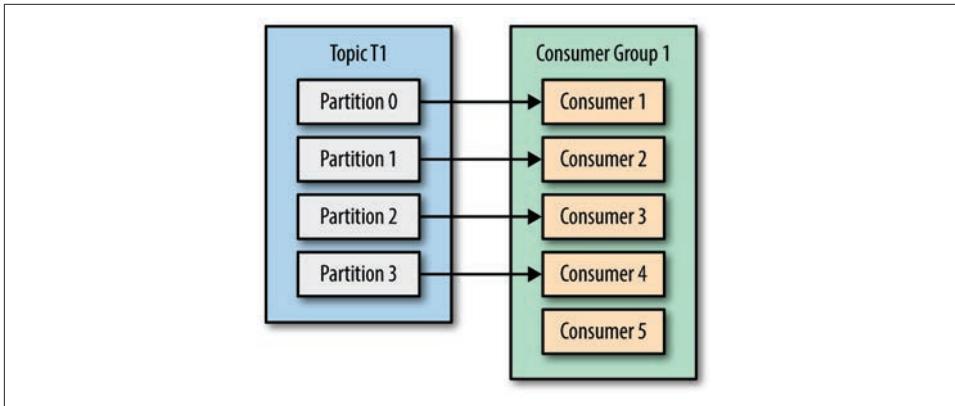


Figure 4-4. More consumer groups than partitions means missed messages

The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data. In these cases, a single consumer can't possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases. Keep in mind that there is no point in adding more consumers than you have partitions in a topic—some of the consumers will just be idle. [Chapter 2](#) includes some suggestions on how to choose the number of partitions in a topic.

In addition to adding consumers in order to scale a single application, it is very common to have multiple applications that need to read data from the same topic. In fact, one of the main design goals in Kafka was to make the data produced to Kafka topics available for many use cases throughout the organization. In those cases, we want each application to get all of the messages, rather than just a subset. To make sure an application gets all the messages in a topic, ensure the application has its own consumer group. Unlike many traditional messaging systems, Kafka scales to a large number of consumers and consumer groups without reducing performance.

In the previous example, if we add a new consumer group G2 with a single consumer, this consumer will get all the messages in topic T1 independent of what G1 is doing. G2 can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for G1, but G2 as a whole will still get all the messages regardless of other consumer groups. See [Figure 4-5](#).

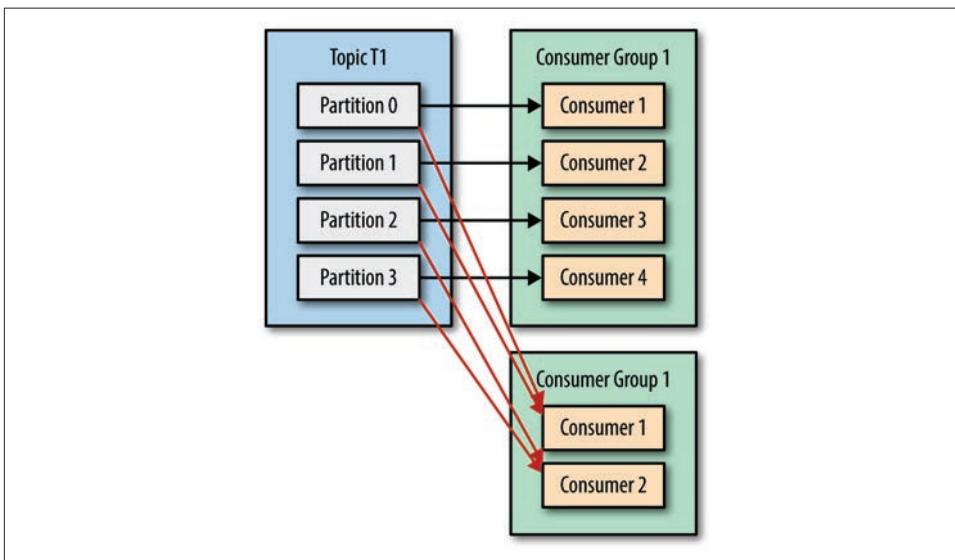


Figure 4-5. Adding a new consumer group ensures no messages are missed

To summarize, you create a new consumer group for each application that needs all the messages from one or more topics. You add consumers to an existing consumer group to scale the reading and processing of messages from the topics, so each additional consumer in a group will only get a subset of the messages.

Consumer Groups and Partition Rebalance

As we saw in the previous section, consumers in a consumer group share ownership of the partitions in the topics they subscribe to. When we add a new consumer to the group, it starts consuming messages from partitions previously consumed by another

consumer. The same thing happens when a consumer shuts down or crashes; it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. Reassignment of partitions to consumers also happen when the topics the consumer group is consuming are modified (e.g., if an administrator adds new partitions).

Moving partition ownership from one consumer to another is called a *rebalance*. Rebalances are important because they provide the consumer group with high availability and scalability (allowing us to easily and safely add and remove consumers), but in the normal course of events they are fairly undesirable. During a rebalance, consumers can't consume messages, so a rebalance is basically a short window of unavailability of the entire consumer group. In addition, when partitions are moved from one consumer to another, the consumer loses its current state; if it was caching any data, it will need to refresh its caches—slowing down the application until the consumer sets up its state again. Throughout this chapter we will discuss how to safely handle rebalances and how to avoid unnecessary ones.

The way consumers maintain membership in a consumer group and ownership of the partitions assigned to them is by sending *heartbeats* to a Kafka broker designated as the *group coordinator* (this broker can be different for different consumer groups). As long as the consumer is sending heartbeats at regular intervals, it is assumed to be alive, well, and processing messages from its partitions. Heartbeats are sent when the consumer polls (i.e., retrieves records) and when it commits records it has consumed.

If the consumer stops sending heartbeats for long enough, its session will time out and the group coordinator will consider it dead and trigger a rebalance. If a consumer crashed and stopped processing messages, it will take the group coordinator a few seconds without heartbeats to decide it is dead and trigger the rebalance. During those seconds, no messages will be processed from the partitions owned by the dead consumer. When closing a consumer cleanly, the consumer will notify the group coordinator that it is leaving, and the group coordinator will trigger a rebalance immediately, reducing the gap in processing. Later in this chapter we will discuss configuration options that control heartbeat frequency and session timeouts and how to set those to match your requirements.

Changes to Heartbeat Behavior in Recent Kafka Versions

In release 0.10.1, the Kafka community introduced a separate heartbeat thread that will send heartbeats in between polls as well. This allows you to separate the heartbeat frequency (and therefore how long it takes for the consumer group to detect that a consumer crashed and is no longer sending heartbeats) from the frequency of polling (which is determined by the time it takes to process the data returned from the brokers). With newer versions of Kafka, you can configure how long the application can go without polling before it will leave the group and trigger a rebalance. This configu-

ration is used to prevent a *livelock*, where the application did not crash but fails to make progress for some reason. This configuration is separate from `session.time.out.ms`, which controls the time it takes to detect a consumer crash and stop sending heartbeats.

The rest of the chapter will discuss some of the challenges with older behaviors and how the programmer can handle them. This chapter includes discussion about how to handle applications that take longer to process records. This is less relevant to readers running Apache Kafka 0.10.1 or later. If you are using a new version and need to handle records that take longer to process, you simply need to tune `max.poll.interval.ms` so it will handle longer delays between polling for new records.



How Does the Process of Assigning Partitions to Brokers Work?

When a consumer wants to join a group, it sends a `JoinGroup` request to the group coordinator. The first consumer to join the group becomes the group *leader*. The leader receives a list of all consumers in the group from the group coordinator (this will include all consumers that sent a heartbeat recently and which are therefore considered alive) and is responsible for assigning a subset of partitions to each consumer. It uses an implementation of `PartitionAssignor` to decide which partitions should be handled by which consumer.

Kafka has two built-in partition assignment policies, which we will discuss in more depth in the configuration section. After deciding on the partition assignment, the consumer leader sends the list of assignments to the `GroupCoordinator`, which sends this information to all the consumers. Each consumer only sees his own assignment—the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

Creating a Kafka Consumer

The first step to start consuming records is to create a `KafkaConsumer` instance. Creating a `KafkaConsumer` is very similar to creating a `KafkaProducer`—you create a Java `Properties` instance with the properties you want to pass to the consumer. We will discuss all the properties in depth later in the chapter. To start we just need to use the three mandatory properties: `bootstrap.servers`, `key.deserializer`, and `value.deserializer`.

The first property, `bootstrap.servers`, is the connection string to a Kafka cluster. It is used the exact same way as in `KafkaProducer` (you can refer to [Chapter 3](#) for

details on how this is defined). The other two properties, `key.deserializer` and `value.deserializer`, are similar to the `serializers` defined for the producer, but rather than specifying classes that turn Java objects to byte arrays, you need to specify classes that can take a byte array and turn it into a Java object.

There is a fourth property, which is not strictly mandatory, but for now we will pretend it is. The property is `group.id` and it specifies the consumer group the `KafkaConsumer` instance belongs to. While it is possible to create consumers that do not belong to any consumer group, this is uncommon, so for most of the chapter we will assume the consumer is part of a group.

The following code snippet shows how to create a `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(props);
```

Most of what you see here should be familiar if you've read [Chapter 3](#) on creating producers. We assume that the records we consume will have `String` objects as both the key and the value of the record. The only new property here is `group.id`, which is the name of the consumer group this consumer belongs to.

Subscribing to Topics

Once we create a consumer, the next step is to subscribe to one or more topics. The `subscribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ①
```

- ① Here we simply create a list with a single element: the topic name `customerCountries`.

It is also possible to call `subscribe` with a regular expression. The expression can match multiple topic names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. Subscribing to multiple topics using a regular expression is most commonly used in applications that replicate data between Kafka and another system.

To subscribe to all test topics, we can call:

```
consumer.subscribe("test.*");
```

The Poll Loop

At the heart of the consumer API is a simple loop for polling the server for more data. Once the consumer subscribes to topics, the poll loop handles all details of coordination, partition rebalances, heartbeats, and data fetching, leaving the developer with a clean API that simply returns available data from the assigned partitions. The main body of a consumer will look as follows:

```
try {
    while (true) { ①
        ConsumerRecords<String, String> records = consumer.poll(100); ②
        for (ConsumerRecord<String, String> record : records) ③
        {
            log.debug("topic = %s, partition = %s, offset = %d,
                      customer = %s, country = %s\n",
                      record.topic(), record.partition(), record.offset(),
                      record.key(), record.value());

            int updatedCount = 1;
            if (custCountryMap.containsValue(record.value())) {
                updatedCount = custCountryMap.get(record.value()) + 1;
            }
            custCountryMap.put(record.value(), updatedCount)

            JSONObject json = new JSONObject(custCountryMap);
            System.out.println(json.toString(4)) ④
        }
    }
} finally {
    consumer.close(); ⑤
}
```

- ❶ This is indeed an infinite loop. Consumers are usually long-running applications that continuously poll Kafka for more data. We will show later in the chapter how to cleanly exit the loop and close the consumer.
- ❷ This is the most important line in the chapter. The same way that sharks must keep moving or they die, consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group to continue consuming. The parameter we pass, `poll()`, is a timeout interval and controls how long `poll()` will block if data is not available in the consumer buffer. If this is set to 0, `poll()` will return immediately; otherwise, it will wait for the specified number of milliseconds for data to arrive from the broker.

- ③ `poll()` returns a list of records. Each record contains the topic and partition the record came from, the offset of the record within the partition, and of course the key and the value of the record. Typically we want to iterate over the list and process the records individually. The `poll()` method takes a timeout parameter. This specifies how long it will take `poll` to return, with or without data. The value is typically driven by application needs for quick responses—how fast do you want to return control to the thread that does the polling?
- ④ Processing usually ends in writing a result in a data store or updating a stored record. Here, the goal is to keep a running count of customers from each county, so we update a hashtable and print the result as JSON. A more realistic example would store the updates result in a data store.
- ⑤ Always `close()` the consumer before exiting. This will close the network connections and sockets. It will also trigger a rebalance immediately rather than wait for the group coordinator to discover that the consumer stopped sending heartbeats and is likely dead, which will take longer and therefore result in a longer period of time in which consumers can't consume messages from a subset of the partitions.

The `poll` loop does a lot more than just get data. The first time you call `poll()` with a new consumer, it is responsible for finding the `GroupCoordinator`, joining the consumer group, and receiving a partition assignment. If a rebalance is triggered, it will be handled inside the `poll` loop as well. And of course the heartbeats that keep consumers alive are sent from within the `poll` loop. For this reason, we try to make sure that whatever processing we do between iterations is fast and efficient.



Thread Safety

You can't have multiple consumers that belong to the same group in one thread and you can't have multiple threads safely use the same consumer. One consumer per thread is the rule. To run multiple consumers in the same group in one application, you will need to run each in its own thread. It is useful to wrap the consumer logic in its own object and then use Java's `ExecutorService` to start multiple threads each with its own consumer. The Confluent blog has a [tutorial](#) that shows how to do just that.

Configuring Consumers

So far we have focused on learning the consumer API, but we've only looked at a few of the configuration properties—just the mandatory `bootstrap.servers`, `group.id`, `key.deserializer`, and `value.deserializer`. All the consumer configuration is documented in Apache Kafka [documentation](#). Most of the parameters have reasonable defaults and do not require modification, but some have implications on the performance and availability of the consumers. Let's take a look at some of the more important properties.

`fetch.min.bytes`

This property allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records. If a broker receives a request for records from a consumer but the new records amount to fewer bytes than `min.fetch.bytes`, the broker will wait until more messages are available before sending the records back to the consumer. This reduces the load on both the consumer and the broker as they have to handle fewer back-and-forth messages in cases where the topics don't have much new activity (or for lower activity hours of the day). You will want to set this parameter higher than the default if the consumer is using too much CPU when there isn't much data available, or reduce load on the brokers when you have large number of consumers.

`fetch.max.wait.ms`

By setting `fetch.min.bytes`, you tell Kafka to wait until it has enough data to send before responding to the consumer. `fetch.max.wait.ms` lets you control how long to wait. By default, Kafka will wait up to 500 ms. This results in up to 500 ms of extra latency in case there is not enough data flowing to the Kafka topic to satisfy the minimum amount of data to return. If you want to limit the potential latency (usually due to SLAs controlling the maximum latency of the application), you can set `fetch.max.wait.ms` to a lower value. If you set `fetch.max.wait.ms` to 100 ms and `fetch.min.bytes` to 1 MB, Kafka will receive a fetch request from the consumer and will respond with data either when it has 1 MB of data to return or after 100 ms, whichever happens first.

`max.partition.fetch.bytes`

This property controls the maximum number of bytes the server will return per partition. The default is 1 MB, which means that when `KafkaConsumer.poll()` returns `ConsumerRecords`, the record object will use at most `max.partition.fetch.bytes` per partition assigned to the consumer. So if a topic has 20 partitions, and you have 5 consumers, each consumer will need to have 4 MB of memory available for `Consumer`

Records. In practice, you will want to allocate more memory as each consumer will need to handle more partitions if other consumers in the group fail. `max.partition.fetch.bytes` must be larger than the largest message a broker will accept (determined by the `max.message.size` property in the broker configuration), or the broker may have messages that the consumer will be unable to consume, in which case the consumer will hang trying to read them. Another important consideration when setting `max.partition.fetch.bytes` is the amount of time it takes the consumer to process data. As you recall, the consumer must call `poll()` frequently enough to avoid session timeout and subsequent rebalance. If the amount of data a single `poll()` returns is very large, it may take the consumer longer to process, which means it will not get to the next iteration of the poll loop in time to avoid a session timeout. If this occurs, the two options are either to lower `max.partition.fetch.bytes` or to increase the session timeout.

session.timeout.ms

The amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 3 seconds. If more than `session.timeout.ms` passes without the consumer sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group. This property is closely related to `heartbeat.interval.ms`. `heartbeat.interval.ms` controls how frequently the `KafkaConsumer poll()` method will send a heartbeat to the group coordinator, whereas `session.timeout.ms` controls how long a consumer can go without sending a heartbeat. Therefore, those two properties are typically modified together—`heartbeat.interval.ms` must be lower than `session.timeout.ms`, and is usually set to one-third of the timeout value. So if `session.timeout.ms` is 3 seconds, `heartbeat.interval.ms` should be 1 second. Setting `session.timeout.ms` lower than the default will allow consumer groups to detect and recover from failure sooner, but may also cause unwanted rebalances as a result of consumers taking longer to complete the poll loop or garbage collection. Setting `session.timeout.ms` higher will reduce the chance of accidental rebalance, but also means it will take longer to detect a real failure.

auto.offset.reset

This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid (usually because the consumer was down for so long that the record with that offset was already aged out of the broker). The default is “latest,” which means that lacking a valid offset, the consumer will start reading from the newest records (records that were written after the consumer started running). The alternative is “earliest,” which

means that lacking a valid offset, the consumer will read all the data in the partition, starting from the very beginning.

enable.auto.commit

We discussed the different options for committing offsets earlier in this chapter. This parameter controls whether the consumer will commit offsets automatically, and defaults to `true`. Set it to `false` if you prefer to control when offsets are committed, which is necessary to minimize duplicates and avoid missing data. If you set `enable.auto.commit` to `true`, then you might also want to control how frequently offsets will be committed using `auto.commit.interval.ms`.

partition.assignment.strategy

We learned that partitions are assigned to consumers in a consumer group. A `PartitionAssignor` is a class that, given consumers and topics they subscribed to, decides which partitions will be assigned to which consumer. By default, Kafka has two assignment strategies:

Range

Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2, and each of the topics has three partitions, then C1 will be assigned partitions 0 and 1 from topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has an uneven number of partitions and the assignment is done for each topic independently, the first consumer ends up with more partitions than the second. This happens whenever Range assignment is used and the number of consumers does not divide the number of partitions in each topic neatly.

RoundRobin

Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described previously used RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1 and partition 1 from topic T2. C2 would have partition 1 from topic T1 and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most 1 partition difference).

The `partition.assignment.strategy` allows you to choose a partition-assignment strategy. The default is `org.apache.kafka.clients.consumer.RangeAssignor`, which implements the Range strategy described above. You can replace it with `org.apache.kafka.clients.consumer.RoundRobinAssignor`. A more advanced option is to implement your own assignment strategy, in which case `partition.assignment.strategy` should point to the name of your class.

client.id

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics, and for quotas.

max.poll.records

This controls the maximum number of records that a single call to `poll()` will return. This is useful to help control the amount of data your application will need to process in the polling loop.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the OS defaults will be used. It can be a good idea to increase those when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

Commits and Offsets

Whenever we call `poll()`, it returns records written to Kafka that consumers in our group have not read yet. This means that we have a way of tracking which records were read by a consumer of the group. As discussed before, one of Kafka's unique characteristics is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their position (offset) in each partition.

We call the action of updating the current position in the partition a **commit**.

How does a consumer commit an offset? It produces a message to Kafka, to a special `__consumer_offsets` topic, with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will trigger a *rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice. See [Figure 4-6](#).

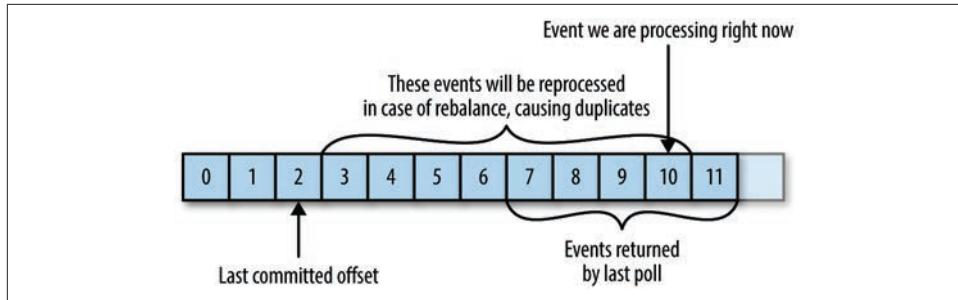


Figure 4-6. Re-processed messages

If the committed offset is larger than the offset of the last message the client actually processed, all messages between the last processed offset and the committed offset will be missed by the consumer group. See Figure 4-7.

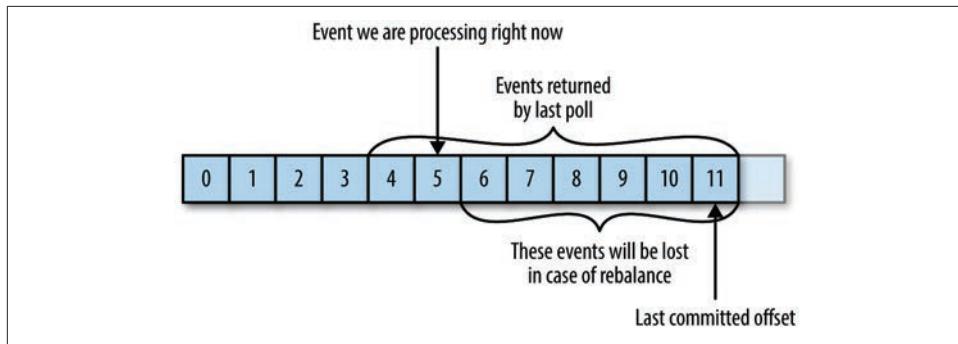


Figure 4-7. Missed messages between offsets

Clearly, managing offsets has a big impact on the client application. The `KafkaConsumer` API provides multiple ways of committing offsets:

Automatic Commit

The easiest way to commit offsets is to allow the consumer to do it for you. If you configure `enable.auto.commit=true`, then every five seconds the consumer will commit the largest offset your client received from `poll()`. The five-second interval is the default and is controlled by setting `auto.commit.interval.ms`. Just like everything else in the consumer, the automatic commits are driven by the poll loop. Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last poll.

Before using this convenient option, however, it is important to understand the consequences.

Consider that, by default, automatic commits occur every five seconds. Suppose that we are three seconds after the most recent commit and a rebalance is triggered. After the rebalancing, all consumers will start consuming from the last offset committed. In this case, the offset is three seconds old, so all the events that arrived in those three seconds will be processed twice. It is possible to configure the commit interval to commit more frequently and reduce the window in which records will be duplicated, but it is impossible to completely eliminate them.

With autocommit enabled, a call to `poll` will always commit the last offset returned by the previous `poll`. It doesn't know which events were actually processed, so it is critical to always process all the events returned by `poll()` before calling `poll()` again. (Just like `poll()`, `close()` also commits offsets automatically.) This is usually not an issue, but pay attention when you handle exceptions or exit the `poll` loop prematurely.

Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

Commit Current Offset

Most developers exercise more control over the time at which offsets are committed—both to eliminate the possibility of missing messages and to reduce the number of messages duplicated during rebalancing. The consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.

By setting `auto.commit.offset=false`, offsets will only be committed when the application explicitly chooses to do so. The simplest and most reliable of the commit APIs is `commitSync()`. This API will commit the latest offset returned by `poll()` and return once the offset is committed, throwing an exception if commit fails for some reason.

It is important to remember that `commitSync()` will commit the latest offset returned by `poll()`, so make sure you call `commitSync()` after you are done processing all the records in the collection, or you risk missing messages as described previously. When rebalance is triggered, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice.

Here is how we would use `commitSync` to commit offsets after we finished processing the latest batch of messages:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset =
        %d, customer = %s, country = %s\n",
        record.topic(), record.partition(),
```

```

        record.offset(), record.key(), record.value()); ①
    }
    try {
        consumer.commitSync(); ②
    } catch (CommitFailedException e) {
        log.error("commit failed", e) ③
    }
}

```

- ➊ Let's assume that by printing the contents of a record, we are done processing it. Your application will likely do a lot more with the records—modify them, enrich them, aggregate them, display them on a dashboard, or notify users of important events. You should determine when you are “done” with a record according to your use case.
- ➋ Once we are done “processing” all the records in the current batch, we call `commitSync` to commit the last offset in the batch, before polling for additional messages.
- ➌ `commitSync` retries committing as long as there is no error that can't be recovered. If this happens, there is not much we can do except log an error.

Asynchronous Commit

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increasing the number of potential duplicates that a rebalance will create.

Another option is the asynchronous commit API. Instead of waiting for the broker to respond to a commit, we just send the request and continue on:

```

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s,
                           offset = %d, customer = %s, country = %s\n",
                           record.topic(), record.partition(), record.offset(),
                           record.key(), record.value());
    }
    consumer.commitAsync(); ①
}

```

- ➊ Commit the last offset and carry on.

The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a nonretryable failure, `commitAsync()` will not retry. The reason

it does not retry is that by the time `commitAsync()` receives a response from the server, there may have been a later commit that was already successful. Imagine that we sent a request to commit offset 2000. There is a temporary communication problem, so the broker never gets the request and therefore never responds. Meanwhile, we processed another batch and successfully committed offset 3000. If `commitAsync()` now retries the previously failed commit, it might succeed in committing offset 2000 *after* offset 3000 was already processed and committed. In the case of a rebalance, this will cause more duplicates.

We mention this complication and the importance of correct order of commits, because `commitAsync()` also gives you an option to pass in a callback that will be triggered when the broker responds. It is common to use the callback to log commit errors or to count them in a metric, but if you want to use the callback for retries, you need to be aware of the problem with commit order:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
                           offset = %d, customer = %s, country = %s\n",
                           record.topic(), record.partition(), record.offset(),
                           record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
                               OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
    }); ①
}
```

- ① We send the commit and carry on, but if the commit fails, the failure and the offsets will be logged.



Retrying Async Commits

A simple pattern to get commit order right for asynchronous retries is to use a monotonically increasing sequence number. Increase the sequence number every time you commit and add the sequence number at the time of the commit to the `commitAsync` callback. When you're getting ready to send a retry, check if the commit sequence number the callback got is equal to the instance variable; if it is, there was no newer commit and it is safe to retry. If the instance sequence number is higher, don't retry because a newer commit was already sent.

Combining Synchronous and Asynchronous Commits

Normally, occasional failures to commit without retrying are not a huge problem because if the problem is temporary, the following commit will be successful. But if we know that this is the last commit before we close the consumer, or before a rebalance, we want to make extra sure that the commit succeeds.

Therefore, a common pattern is to combine `commitAsync()` with `commitSync()` just before shutdown. Here is how it works (we will discuss how to commit just before rebalance when we get to the section about rebalance listeners):

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
customer = %s, country = %s\n",
record.topic(), record.partition(),
record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ①
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ②
    } finally {
        consumer.close();
    }
}
```

- ① While everything is fine, we use `commitAsync`. It is faster, and if one commit fails, the next commit will serve as a retry.
- ② But if we are closing, there is no “next commit.” We call `commitSync()`, because it will retry until it succeeds or suffers unrecoverable failure.

Commit Specified Offset

Committing the latest offset only allows you to commit as often as you finish processing batches. But what if you want to commit more frequently than that? What if `poll()` returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those rows again if a rebalance occurs? You can’t just call `commitSync()` or `commitAsync()`—this will commit the last offset returned, which you didn’t get to process yet.

Fortunately, the consumer API allows you to call `commitSync()` and `commitAsync()` and pass a map of partitions and offsets that you wish to commit. If you are in the middle of processing a batch of records, and the last message you got from partition 3 in topic “customers” has offset 5000, you can call `commitSync()` to commit offset 5000 for partition 3 in topic “customers.” Since your consumer may be consuming more than a single partition, you will need to track offsets on all of them, which adds complexity to your code.

Here is what a commit of specific offsets looks like:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ①
int count = 0;

....
```



```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d,
customer = %s, country = %s\n",
                           record.topic(), record.partition(), record.offset(),
                           record.key(), record.value()); ②
        currentOffsets.put(new TopicPartition(record.topic(),
                                               record.partition()), new
                           OffsetAndMetadata(record.offset() + 1, "no metadata")); ③
        if (count % 1000 == 0) ④
            consumer.commitAsync(currentOffsets, null); ⑤
        count++;
    }
}
```

- ① This is the map we will use to manually track offsets.
- ② Remember, `println` is a stand-in for whatever processing you do for the records you consume.
- ③ After reading each record, we update the offsets map with the offset of the next message we expect to process. This is where we'll start reading next time we start.
- ④ Here, we decide to commit current offsets every 1,000 records. In your application, you can commit based on time or perhaps content of the records.
- ⑤ I chose to call `commitAsync()`, but `commitSync()` is also completely valid here. Of course, when committing specific offsets you still need to perform all the error handling we've seen in previous sections.

Rebalance Listeners

As we mentioned in the previous section about committing offsets, a consumer will want to do some cleanup work before exiting and also before partition rebalancing.

If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed. If your consumer maintained a buffer with events that it only processes occasionally (e.g., the `currentRecords` map we used when explaining `pause()` functionality), you will want to process the events you accumulated before losing ownership of the partition. Perhaps you also need to close file handles, database connections, and such.

The consumer API allows you to run your own code when partitions are added or removed from the consumer. You do this by passing a `ConsumerRebalanceListener` when calling the `subscribe()` method we discussed previously. `ConsumerRebalanceListener` has two methods you can implement:

```
public void onPartitionsRevoked(Collection<TopicPartition> partitions)
    Called before the rebalancing starts and after the consumer stopped consuming
    messages. This is where you want to commit offsets, so whoever gets this parti-
    tion next will know where to start.
```

```
public void onPartitionsAssigned(Collection<TopicPartition> partitions)
    Called after partitions have been reassigned to the broker, but before the con-
    sumer starts consuming messages.
```

This example will show how to use `onPartitionsRevoked()` to commit offsets before losing ownership of a partition. In the next section we will show a more involved example that also demonstrates the use of `onPartitionsAssigned()`:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>();

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) { ❷
    }

    public void onPartitionsRevoked(Collection<TopicPartition>
        partitions) {
        System.out.println("Lost partitions in rebalance.
            Committing current
            offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
    }
}

try {
```

```

consumer.subscribe(topics, new HandleRebalance()); ④

while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d,
                           customer = %s, country = %s\n",
                           record.topic(), record.partition(), record.offset(),
                           record.key(), record.value());
        currentOffsets.put(new TopicPartition(record.topic(),
                                               record.partition()), new
                           OffsetAndMetadata(record.offset() + 1, "no metadata"));
    }
    consumer.commitAsync(currentOffsets, null);
}
} catch (WakeupException e) {
    // ignore, we're closing
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
}

```

- ➊ We start by implementing a `ConsumerRebalanceListener`.
- ➋ In this example we don't need to do anything when we get a new partition; we'll just start consuming messages.
- ➌ However, when we are about to lose a partition due to rebalancing, we need to commit offsets. Note that we are committing the latest offsets we've processed, not the latest offsets in the batch we are still processing. This is because a partition could get revoked while we are still in the middle of a batch. We are committing offsets for all partitions, not just the partitions we are about to lose—because the offsets are for events that were already processed, there is no harm in that. And we are using `commitSync()` to make sure the offsets are committed before the rebalance proceeds.
- ➍ The most important part: pass the `ConsumerRebalanceListener` to the `subscribe()` method so it will get invoked by the consumer.

Consuming Records with Specific Offsets

So far we've seen how to use `poll()` to start consuming messages from the last committed offset in each partition and to proceed in processing all messages in sequence. However, sometimes you want to start reading at a different offset.

If you want to start reading all messages from the beginning of the partition, or you want to skip all the way to the end of the partition and start consuming only new messages, there are APIs specifically for that: `seekToBeginning(TopicPartition tp)` and `seekToEnd(TopicPartition tp)`.

However, the Kafka API also lets you seek a specific offset. This ability can be used in a variety of ways; for example, to go back a few messages or skip ahead a few messages (perhaps a time-sensitive application that is falling behind will want to skip ahead to more relevant messages). The most exciting use case for this ability is when offsets are stored in a system other than Kafka.

Think about this common scenario: Your application is reading events from Kafka (perhaps a clickstream of users in a website), processes the data (perhaps remove records that indicate clicks from automated programs rather than users), and then stores the results in a database, NoSQL store, or Hadoop. Suppose that we really don't want to lose any data, nor do we want to store the same results in the database twice.

In these cases, the consumer loop may look a bit like this:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        currentOffsets.put(new TopicPartition(record.topic(),
            record.partition(),
            record.offset());
        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(currentOffsets);
    }
}
```

In this example, we are very paranoid, so we commit offsets after processing each record. However, there is still a chance that our application will crash after the record was stored in the database but before we committed offsets, causing the record to be processed again and the database to contain duplicates.

This could be avoided if there was a way to store both the record and the offset in one atomic action. Either both the record and the offset are committed, or neither of them are committed. As long as the records are written to a database and the offsets to Kafka, this is impossible.

But what if we wrote both the record and the offset to the database, in one transaction? Then we'll know that either we are done with the record and the offset is committed or we are not and the record will be reprocessed.

Now the only problem is if the record is stored in a database and not in Kafka, how will our consumer know where to start reading when it is assigned a partition? This is exactly what `seek()` can be used for. When the consumer starts or when new partitions are assigned, it can look up the offset in the database and `seek()` to that location.

Here is a skeleton example of how this may work. We use `ConsumerRebalanceListener` and `seek()` to make sure we start processing at the offsets stored in the database:

```
public class SaveOffsetsOnRebalance implements
    ConsumerRebalanceListener {

    public void onPartitionsRevoked(Collection<TopicPartition>
        partitions) {
        commitDBTransaction(); ❶
    }

    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) {
        for(TopicPartition partition: partitions)
            consumer.seek(partition, getOffsetFromDB(partition)); ❷
    }
}

consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);

for (TopicPartition partition: consumer.assignment())
    consumer.seek(partition, getOffsetFromDB(partition)); ❸

while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        processRecord(record);
        storeRecordInDB(record);
        storeOffsetInDB(record.topic(), record.partition(),
            record.offset()); ❹
    }
    commitDBTransaction();
}
```

- ❶ We use an imaginary method here to commit the transaction in the database. The idea here is that the database records and offsets will be inserted to the database as we process the records, and we just need to commit the transactions when we are about to lose the partition to make sure this information is persisted.
- ❷ We also have an imaginary method to fetch the offsets from the database, and then we `seek()` to those records when we get ownership of new partitions.
- ❸ When the consumer first starts, after we subscribe to topics, we call `poll()` once to make sure we join a consumer group and get assigned partitions, and then we immediately `seek()` to the correct offset in the partitions we are assigned to. Keep in mind that `seek()` only updates the position we are consuming from, so the next `poll()` will fetch the right messages. If there was an error in `seek()` (e.g., the offset does not exist), the exception will be thrown by `poll()`.
- ❹ Another imaginary method: this time we update a table storing the offsets in our database. Here we assume that updating records is fast, so we do an update on every record, but commits are slow, so we only commit at the end of the batch. However, this can be optimized in different ways.

There are many different ways to implement exactly-once semantics by storing offsets and data in an external store, but all of them will need to use the `ConsumerRebalanceListener` and `seek()` to make sure offsets are stored in time and that the consumer starts reading messages from the correct location.

But How Do We Exit?

Earlier in this chapter, when we discussed the poll loop, I told you not to worry about the fact that the consumer polls in an infinite loop and that we would discuss how to exit the loop cleanly. So, let's discuss how to exit cleanly.

When you decide to exit the poll loop, you will need another thread to call `consumer.wakeup()`. If you are running the consumer loop in the main thread, this can be done from `ShutdownHook`. Note that `consumer.wakeup()` is the only consumer method that is safe to call from a different thread. Calling `wakeup` will cause `poll()` to exit with `WakeUpException`, or if `consumer.wakeup()` was called while the thread was not waiting on `poll`, the exception will be thrown on the next iteration when `poll()` is called. The `WakeUpException` doesn't need to be handled, but before exiting the thread, you must call `consumer.close()`. Closing the consumer will commit offsets if needed and will send the group coordinator a message that the consumer is leaving the group. The consumer coordinator will trigger rebalancing immediately

and you won't need to wait for the session to time out before partitions from the consumer you are closing will be assigned to another consumer in the group.

Here is what the exit code will look like if the consumer is running in the main application thread. This example is a bit truncated, but you can view the full example at <http://bit.ly/2u47e9A>.

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup(); ❶
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
```

...

```
try {
    // looping until ctrl-c, the shutdown hook will
    // cleanup on exit
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(1000);
        System.out.println(System.currentTimeMillis() + "
            -- waiting for data...");
        for (ConsumerRecord<String, String> record :
            records) {
            System.out.printf("offset = %d, key = %s,
                value = %s\n",
                record.offset(), record.key(),
                record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Committing offset at
                position:" +
                consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
    // ignore for shutdown ❷
} finally {
    consumer.close(); ❸
    System.out.println("Closed consumer and we are done");
}
```

- ① ShutdownHook runs in a separate thread, so the only safe action we can take is to call wakeup to break out of the poll loop.
- ② Another thread calling wakeup will cause poll to throw a `WakeupException`. You'll want to catch the exception to make sure your application doesn't exit unexpectedly, but there is no need to do anything with it.
- ③ Before exiting the consumer, make sure you close it cleanly.

Deserializers

As discussed in the previous chapter, Kafka producers require *serializers* to convert objects into byte arrays that are then sent to Kafka. Similarly, Kafka consumers require *deserializers* to convert byte arrays received from Kafka into Java objects. In previous examples, we just assumed that both the key and the value of each message are strings and we used the default `StringDeserializer` in the consumer configuration.

In [Chapter 3](#) about the Kafka producer, we saw how to serialize custom types and how to use Avro and `AvroSerializers` to generate Avro objects from schema definitions and then serialize them when producing messages to Kafka. We will now look at how to create custom deserializers for your own objects and how to use Avro and its deserializers.

It should be obvious that the serializer used to produce events to Kafka must match the deserializer that will be used when consuming events. Serializing with `IntSerializer` and then deserializing with `StringDeserializer` will not end well. This means that as a developer you need to keep track of which serializers were used to write into each topic, and make sure each topic only contains data that the deserializers you use can interpret. This is one of the benefits of using Avro and the Schema Repository for serializing and deserializing—the `AvroSerializer` can make sure that all the data written to a specific topic is compatible with the schema of the topic, which means it can be deserialized with the matching deserializer and schema. Any errors in compatibility—on the producer or the consumer side—will be caught easily with an appropriate error message, which means you will not need to try to debug byte arrays for serialization errors.

We will start by quickly showing how to write a custom deserializer, even though this is the less common method, and then we will move on to an example of how to use Avro to deserialize message keys and values.

Custom deserializers

Let's take the same custom object we serialized in [Chapter 3](#), and write a deserializer for it:

```
public class Customer {  
    private int customerID;  
    private String customerName;  
  
    public Customer(int ID, String name) {  
        this.customerID = ID;  
        this.customerName = name;  
    }  
  
    public int getID() {  
        return customerID;  
    }  
  
    public String getName() {  
        return customerName;  
    }  
}
```

The custom deserializer will look as follows:

```
import org.apache.kafka.common.errors.SerializationException;  
  
import java.nio.ByteBuffer;  
import java.util.Map;  
  
public class CustomerDeserializer implements  
    Deserializer<Customer> {❶  
  
    @Override  
    public void configure(Map configs, boolean isKey) {  
        // nothing to configure  
    }  
  
    @Override  
    public Customer deserialize(String topic, byte[] data) {  
  
        int id;  
        int nameSize;  
        String name;  
  
        try {  
            if (data == null)  
                return null;  
            if (data.length < 8)  
                throw new SerializationException("Size of data received by  
                    IntegerDeserializer is shorter than expected");  
  
            ByteBuffer buffer = ByteBuffer.wrap(data);  
        }
```

```

        id = buffer.getInt();
        String nameSize = buffer.getInt();

        byte[] nameBytes = new Array[Byte](nameSize);
        buffer.get(nameBytes);
        name = new String(nameBytes, 'UTF-8');

        return new Customer(id, name); ②

    } catch (Exception e) {
        throw new SerializationException("Error when serializing
            Customer
            to byte[] " + e);
    }
}

@Override
public void close() {
    // nothing to close
}
}

```

- ① The consumer also needs the implementation of the `Customer` class, and both the class and the serializer need to match on the producing and consuming applications. In a large organization with many consumers and producers sharing access to the data, this can become challenging.
- ② We are just reversing the logic of the serializer here—we get the customer ID and name out of the byte array and use them to construct the object we need.

The consumer code that uses this serializer will look similar to this example:

```

Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
        "org.apache.kafka.common.serialization.CustomerDeserializer");

KafkaConsumer<String, Customer> consumer =
    new KafkaConsumer<>(props);

consumer.subscribe("customerCountries")

while (true) {
    ConsumerRecords<String, Customer> records =
        consumer.poll(100);
    for (ConsumerRecord<String, Customer> record : records)
    {
        System.out.println("current customer Id: " +

```

```

        record.value().getId() + " and
        current customer name: " + record.value().getName());
    }
}

```

Again, it is important to note that implementing a custom serializer and deserializer is not recommended. It tightly couples producers and consumers and is fragile and error-prone. A better solution would be to use a standard message format such as JSON, Thrift, Protobuf, or Avro. We'll now see how to use Avro deserializers with the Kafka consumer. For background on Apache Avro, its schemas, and schema-compatibility capabilities, refer back to [Chapter 3](#).

Using Avro deserialization with Kafka consumer

Let's assume we are using the implementation of the `Customer` class in Avro that was shown in [Chapter 3](#). In order to consume those objects from Kafka, you want to implement a consuming application similar to this:

```

Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.serializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
        "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer consumer = new
    KafkaConsumer(createConsumerConfig(brokers, groupId, url));
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records =
        consumer.poll(1000); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
                           record.value().getName()); ❹
    }
    consumer.commitSync();
}

```

- ❶ We use `KafkaAvroDeserializer` to deserialize the Avro messages.
- ❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas. This way the consumer can use the schema that was registered by the producer to deserialize the message.

- ③ We specify the generated class, `Customer`, as the type for the record value.
- ④ `record.value()` is a `Customer` instance and we can use it accordingly.

Standalone Consumer: Why and How to Use a Consumer Without a Group

So far, we have discussed consumer groups, which are where partitions are assigned automatically to consumers and are rebalanced automatically when consumers are added or removed from the group. Typically, this behavior is just what you want, but in some cases you want something much simpler. Sometimes you know you have a single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic. In this case, there is no reason for groups or rebalances—just assign the consumer-specific topic and/or partitions, consume messages, and commit offsets on occasion.

When you know exactly which partitions the consumer should read, you don't *subscribe* to a topic—instead, you *assign* yourself a few partitions. A consumer can either subscribe to topics (and be part of a consumer group), or assign itself partitions, but not both at the same time.

Here is an example of how a consumer can assign itself all partitions of a specific topic and consume from them:

```

List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ①

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ②

    while (true) {
        ConsumerRecords<String, String> records =
            consumer.poll(1000);

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}

```

- ❶ We start by asking the cluster for the partitions available in the topic. If you only plan on consuming a specific partition, you can skip this part.
- ❷ Once we know which partitions we want, we call `assign()` with the list.

Other than the lack of rebalances and the need to manually find the partitions, everything else is business as usual. Keep in mind that if someone adds new partitions to the topic, the consumer will not be notified. You will need to handle this by checking `consumer.partitionsFor()` periodically or simply by bouncing the application whenever partitions are added.

Older Consumer APIs

In this chapter we discussed the Java `KafkaConsumer` client that is part of the `org.apache.kafka.clients` package. At the time of writing, Apache Kafka still has two older clients written in Scala that are part of the `kafka.consumer` package, which is part of the core Kafka module. These consumers are called `SimpleConsumer` (which is not very simple). `SimpleConsumer` is a thin wrapper around the Kafka APIs that allows you to consume from specific partitions and offsets. The other old API is called high-level consumer or `ZookeeperConsumerConnector`. The high-level consumer is somewhat similar to the current consumer in that it has consumer groups and it rebalances partitions, but it uses Zookeeper to manage consumer groups and does not give you the same control over commits and rebalances as we have now.

Because the current consumer supports both behaviors and provides much more reliability and control to the developer, we will not discuss the older APIs. If you are interested in using them, please think twice and then refer to Apache Kafka documentation to learn more.

Summary

We started this chapter with an in-depth explanation of Kafka's consumer groups and the way they allow multiple consumers to share the work of reading events from topics. We followed the theoretical discussion with a practical example of a consumer subscribing to a topic and continuously reading events. We then looked into the most important consumer configuration parameters and how they affect consumer behavior. We dedicated a large part of the chapter to discussing offsets and how consumers keep track of them. Understanding how consumers commit offsets is critical when writing reliable consumers, so we took time to explain the different ways this can be done. We then discussed additional parts of the consumer APIs, handling rebalances and closing the consumer.

We concluded by discussing the deserializers used by consumers to turn bytes stored in Kafka into Java objects that the applications can process. We discussed Avro deserializers in some detail, even though they are just one type of deserializer you can use, because these are most commonly used with Kafka.

Now that you know how to produce and consume events with Kafka, the next chapter explains some of the internals of a Kafka implementation.

CHAPTER 11

Stream Processing

Kafka was traditionally seen as a powerful message bus, capable of delivering streams of events but without processing or transformation capabilities. Kafka's reliable stream delivery capabilities make it a perfect source of data for stream-processing systems. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza, and many more stream-processing systems were built with Kafka often being their only reliable data source.

Industry analysts sometimes claim that all those stream-processing systems are just like the complex event processing (CEP) systems that have been around for 20 years. We think stream processing became more popular because it was created after Kafka and therefore could use Kafka as a reliable source of event streams to process. With the increased popularity of Apache Kafka, first as a simple message bus and later as a data integration system, many companies had a system containing many streams of interesting data, stored for long amounts of time and perfectly ordered, just waiting for some stream-processing framework to show up and process them. In other words, in the same way that data processing was significantly more difficult before databases were invented, stream processing was held back by lack of a stream-processing platform.

Starting from version 0.10.0, Kafka does more than provide a reliable source of data streams to every popular stream-processing framework. Now Kafka includes a powerful stream-processing library as part of its collection of client libraries. This allows developers to consume, process, and produce events in their own apps, without relying on an external processing framework.

We'll begin the chapter by explaining what we mean by stream processing (since this term is frequently misunderstood), then discuss some of the basic concepts of stream processing and the design patterns that are common to all stream-processing systems. We'll then dive into Apache Kafka's stream-processing library—its goals and

architecture. We'll give a small example of how to use Kafka Streams to calculate a moving average of stock prices. We'll then discuss other examples for good stream-processing use cases and finish off the chapter by providing a few criteria you can use when choosing which stream-processing framework (if any) to use with Apache Kafka. This chapter is intended as a brief introduction to stream processing and will not cover every Kafka Streams feature or attempt to discuss and compare every stream-processing framework in existence—those topics deserve entire books on their own, possibly several.

What Is Stream Processing?

There is a lot of confusion about what stream processing means. Many definitions mix up implementation details, performance requirements, data models, and many other aspects of software engineering. I've seen the same thing play out in the world of relational databases—the abstract definitions of the relational model are getting forever entangled in the implementation details and specific limitations of the popular database engines.

The world of stream processing is still evolving, and just because a specific popular implementation does things in specific ways or has specific limitations doesn't mean that those details are an inherent part of processing streams of data.

Let's start at the beginning: What is a data stream (also called an *event stream* or *streaming data*)? First and foremost, a *data stream* is an abstraction representing an unbounded dataset. *Unbounded* means infinite and ever growing. The dataset is unbounded because over time, new records keep arriving. This definition is used by [Google](#), [Amazon](#), and pretty much everyone else.

Note that this simple model (a stream of events) can be used to represent pretty much every business activity we care to analyze. We can look at a stream of credit card transactions, stock trades, package deliveries, network events going through a switch, events reported by sensors in manufacturing equipment, emails sent, moves in a game, etc. The list of examples is endless because pretty much everything can be seen as a sequence of events.

There are few other attributes of event streams model, in addition to their unbounded nature:

Event streams are ordered

There is an inherent notion of which events occur before or after other events. This is clearest when looking at financial events. A sequence in which I first put money in my account and later spend the money is very different from a sequence at which I first spend the money and later cover my debt by depositing money back. The latter will incur overdraft charges while the former will not. Note that this is one of the differences between an event stream and a database

table—records in a table are always considered unordered and the “order by” clause of SQL is not part of the relational model; it was added to assist in reporting.

Immutable data records

Events, once occurred, can never be modified. A financial transaction that is cancelled does not disappear. Instead, an additional event is written to the stream, recording a cancellation of previous transaction. When a customer returns merchandise to a shop, we don’t delete the fact that the merchandise was sold to him earlier, rather we record the return as an additional event. This is another difference between a data stream and a database table—we can delete or update records in a table, but those are all additional transactions that occur in the database, and as such can be recorded in a stream of events that records all transactions. If you are familiar with binlogs, WALs, or redo logs in databases you can see that if we insert a record into a table and later delete it, the table will no longer contain the record, but the redo log will contain two transactions—the insert and the delete.

Event streams are replayable

This is a desirable property. While it is easy to imagine nonreplayable streams (TCP packets streaming through a socket are generally nonreplayable), for most business applications, it is critical to be able to replay a raw stream of events that occurred months (and sometimes years) earlier. This is required in order to correct errors, try new methods of analysis, or perform audits. This is the reason we believe Kafka made stream processing so successful in modern businesses—it allows capturing and replaying a stream of events. Without this capability, stream processing would not be more than a lab toy for data scientists.

It is worth noting that neither the definition of event streams nor the attributes we later listed say anything about the data contained in the events or the number of events per second. The data differs from system to system—events can be tiny (sometimes only a few bytes) or very large (XML messages with many headers); they can also be completely unstructured, key-value pairs, semi-structured JSON, or structured Avro or Protobuf messages. While it is often assumed that data streams are “big data” and involve millions of events per second, the same techniques we’ll discuss apply equally well (and often better) to smaller streams of events with only a few events per second or minute.

Now that we know what event streams are, it’s time to make sure we understand stream processing. Stream processing refers to the ongoing processing of one or more event streams. Stream processing is a programming paradigm—just like request-response and batch processing. Let’s look at how different programming paradigms compare to get a better understanding of how stream processing fits into software architectures:

Request-response

This is the lowest latency paradigm, with response times ranging from submilliseconds to a few milliseconds, usually with the expectation that response times will be highly consistent. The mode of processing is usually blocking—an app sends a request and waits for the processing system to respond. In the database world, this paradigm is known as *online transaction processing* (OLTP). Point-of-sale systems, credit card processing, and time-tracking systems typically work in this paradigm.

Batch processing

This is the high-latency/high-throughput option. The processing system wakes up at set times—every day at 2:00 A.M., every hour on the hour, etc. It reads all required input (either all data available since last execution, all data from beginning of month, etc.), writes all required output, and goes away until the next time it is scheduled to run. Processing times range from minutes to hours and users expect to read stale data when they are looking at results. In the database world, these are the data warehouse and business intelligence systems—data is loaded in huge batches once a day, reports are generated, and users look at the same reports until the next data load occurs. This paradigm often has great efficiency and economy of scale, but in recent years, businesses need the data available in shorter timeframes in order to make decision-making more timely and efficient. This puts huge pressure on systems that were written to exploit economy of scale—not to provide low-latency reporting.

Stream processing

This is a contentious and nonblocking option. Filling the gap between the request-response world where we wait for events that take two milliseconds to process and the batch processing world where data is processed once a day and takes eight hours to complete. Most business processes don't require an immediate response within milliseconds but can't wait for the next day either. Most business processes happen continuously, and as long as the business reports are updated continuously and the line of business apps can continuously respond, the processing can proceed without anyone waiting for a specific response within milliseconds. Business processes like alerting on suspicious credit transactions or network activity, adjusting prices in real-time based on supply and demand, or tracking deliveries of packages are all natural fit for continuous but nonblocking processing.

It is important to note that the definition doesn't mandate any specific framework, API, or feature. As long as you are continuously reading data from an unbounded dataset, doing something to it, and emitting output, you are doing stream processing. But the processing has to be continuous and ongoing. A process that starts every day at 2:00 A.M., reads 500 records from the stream, outputs a result, and goes away doesn't quite cut it as far as stream processing goes.

Stream-Processing Concepts

Stream processing is very similar to any type of data processing—you write code that receives data, does something with the data—a few transformations, aggregates, enrichments, etc.—and then place the result somewhere. However, there are some key concepts that are unique to stream processing and often cause confusion when someone who has data processing experience first attempts to write stream-processing applications. Let’s take a look at a few of those concepts.

Time

Time is probably the most important concept in stream processing and often the most confusing. For an idea of how complex time can get when discussing distributed systems, we recommend Justin Sheehy’s excellent “[There is No Now](#)” paper. In the context of stream processing, having a common notion of time is critical because most stream applications perform operations on time windows. For example, our stream application might calculate a moving five-minute average of stock prices. In that case, we need to know what to do when one of our producers goes offline for two hours due to network issues and returns with two hours worth of data—most of the data will be relevant for five-minute time windows that have long passed and for which the result was already calculated and stored.

Stream-processing systems typically refer to the following notions of time:

Event time

This is the time the events we are tracking occurred and the record was created—the time a measurement was taken, an item was sold at a shop, a user viewed a page on our website, etc. In versions 0.10.0 and later, Kafka automatically adds the current time to producer records at the time they are created. If this does not match your application’s notion of *event time*, such as in cases where the Kafka record is created based on a database record some time after the event occurred, you should add the event time as a field in the record itself. Event time is usually the time that matters most when processing stream data.

Log append time

This is the time the event arrived to the Kafka broker and was stored there. In versions 0.10.0 and higher, Kafka brokers will automatically add this time to records they receive if Kafka is configured to do so or if the records arrive from older producers and contain no timestamps. This notion of time is typically less relevant for stream processing, since we are usually interested in the times the events occurred. For example, if we calculate number of devices produced per day, we want to count devices that were actually produced on that day, even if there were network issues and the event only arrived to Kafka the following day. However, in cases where the real event time was not recorded, log append time

can still be used consistently because it does not change after the record was created.

Processing time

This is the time at which a stream-processing application received the event in order to perform some calculation. This time can be milliseconds, hours, or days after the event occurred. This notion of time assigns different timestamps to the same event depending on exactly when each stream processing application happened to read the event. It can even differ for two threads in the same application! Therefore, this notion of time is highly unreliable and best avoided.



Mind the Time Zone

When working with time, it is important to be mindful of time zones. The entire data pipeline should standardize on a single time zone; otherwise, results of stream operations will be confusing and often meaningless. If you must handle data streams with different time zones, you need to make sure you can convert events to a single time zone before performing operations on time windows. Often this means storing the time zone in the record itself.

State

As long as you only need to process each event individually, stream processing is a very simple activity. For example, if all you need to do is read a stream of online shopping transactions from Kafka, find the transactions over \$10,000 and email the relevant salesperson, you can probably write this in just few lines of code using a Kafka consumer and SMTP library.

Stream processing becomes really interesting when you have operations that involve multiple events: counting the number of events by type, moving averages, joining two streams to create an enriched stream of information, etc. In those cases, it is not enough to look at each event by itself; you need to keep track of more information—how many events of each type did we see this hour, all events that require joining, sums, averages, etc. We call the information that is stored between events a *state*.

It is often tempting to store the state in variables that are local to the stream-processing app, such as a simple hash-table to store moving counts. In fact, we did just that in many examples in this book. However, this is not a reliable approach for managing state in stream processing because when the stream-processing application is stopped, the state is lost, which changes the results. This is usually not the desired outcome, so care should be taken to persist the most recent state and recover it when starting the application.

Stream processing refers to several types of state:

Local or internal state

State that is accessible only by a specific instance of the stream-processing application. This state is usually maintained and managed with an embedded, in-memory database running within the application. The advantage of local state is that it is extremely fast. The disadvantage is that you are limited to the amount of memory available. As a result, many of the design patterns in stream processing focus on ways to partition the data into substreams that can be processed using a limited amount of local state.

External state

State that is maintained in an external datastore, often a NoSQL system like Cassandra. The advantages of an external state are its virtually unlimited size and the fact that it can be accessed from multiple instances of the application or even from different applications. The downside is the extra latency and complexity introduced with an additional system. Most stream-processing apps try to avoid having to deal with an external store, or at least limit the latency overhead by caching information in the local state and communicating with the external store as rarely as possible. This usually introduces challenges with maintaining consistency between the internal and external state.

Stream-Table Duality

We are all familiar with database tables. A table is a collection of records, each identified by its primary key and containing a set of attributes as defined by a schema. Table records are mutable (i.e., tables allow update and delete operations). Querying a table allows checking the state of the data at a specific point in time. For example, by querying the CUSTOMERS_CONTACTS table in a database, we expect to find current contact details for all our customers. Unless the table was specifically designed to include history, we will not find their past contacts in the table.

Unlike tables, streams contain a history of changes. Streams are a string of events wherein each event caused a change. A table contains a current state of the world, which is the result of many changes. From this description, it is clear that streams and tables are two sides of the same coin—the world always changes, and sometimes we are interested in the events that caused those changes, whereas other times we are interested in the current state of the world. Systems that allow you to transition back and forth between the two ways of looking at data are more powerful than systems that support just one.

In order to convert a table to a stream, we need to capture the changes that modify the table. Take all those `insert`, `update`, and `delete` events and store them in a stream. Most databases offer change data capture (CDC) solutions for capturing these changes and there are many Kafka connectors that can pipe those changes into Kafka where they will be available for stream processing.

In order to convert a stream to a table, we need to apply all the changes that the stream contains. This is also called *materializing* the stream. We create a table, either in memory, in an internal state store, or in an external database, and start going over all the events in the stream from beginning to end, changing the state as we go. When we finish, we have a table representing a state at a specific time that we can use.

Suppose we have a store selling shoes. A stream representation of our retail activity can be a stream of events:

“Shipment arrived with red, blue, and green shoes”

“Blue shoes sold”

“Red shoes sold”

“Blue shoes returned”

“Green shoes sold”

If we want to know what our inventory contains right now or how much money we made until now, we need to materialize the view. Figure 11-1 shows that we currently have blue and yellow shoes and \$170 in the bank. If we want to know how busy the store is, we can look at the entire stream and see that there were five transactions. We may also want to investigate why the blue shoes were returned.

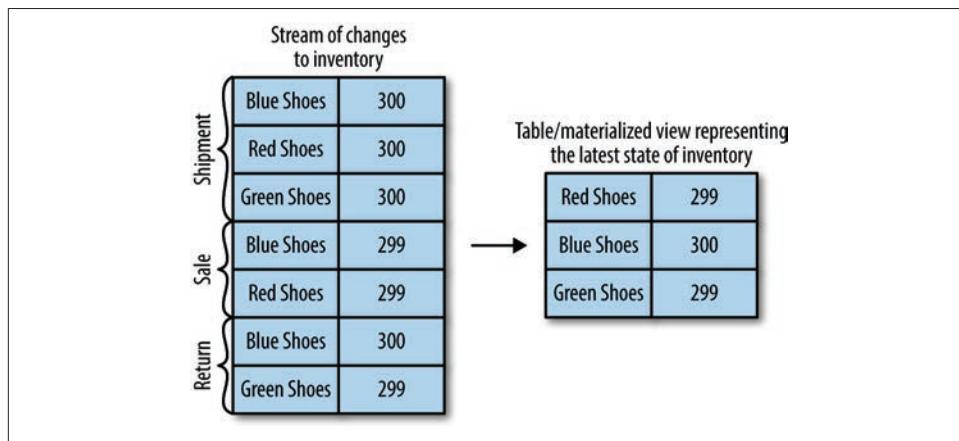


Figure 11-1. Materializing inventory changes

Time Windows

Most operations on streams are windowed operations—operating on slices of time: moving averages, top products sold this week, 99th percentile load on the system, etc. Join operations on two streams are also windowed—we join events that occurred at the same slice of time. Very few people stop and think about the type of window they

want for their operations. For example, when calculating moving averages, we want to know:

- Size of the window: do we want to calculate the average of all events in every five-minute window? Every 15-minute window? Or the entire day? Larger windows are smoother but they lag more—if price increases, it will take longer to notice than with a smaller window.
- How often the window moves (*advance interval*): five-minute averages can update every minute, second, or every time there is a new event. When the *advance interval* is equal to the window size, this is sometimes called a *tumbling window*. When the window moves on every record, this is sometimes called a *sliding window*.
- How long the window remains updatable: our five-minute moving average calculated the average for 00:00-00:05 window. Now an hour later, we are getting a few more results with their *event time* showing 00:02. Do we update the result for the 00:00-00:05 period? Or do we let bygones be bygones? Ideally, we'll be able to define a certain time period during which events will get added to their respective time-slice. For example, if the events were up to four hours late, we should recalculate the results and update. If events arrive later than that, we can ignore them.

Windows can be aligned to clock time—i.e., a five-minute window that moves every minute will have the first slice as 00:00-00:05 and the second as 00:01-00:06. Or it can be unaligned and simply start whenever the app started and then the first slice can be 03:17-03:22. Sliding windows are never aligned because they move whenever there is a new record. See [Figure 11-2](#) for the difference between two types of these windows.

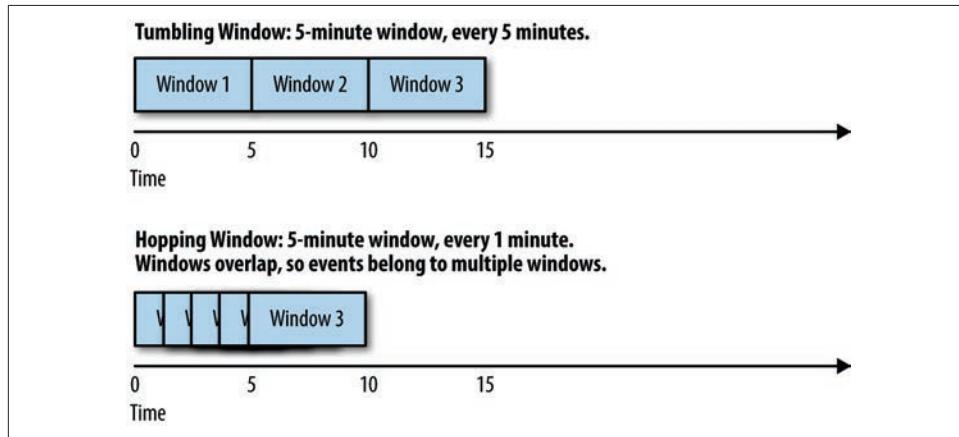


Figure 11-2. Tumbling window versus Hopping window

Stream-Processing Design Patterns

Every stream-processing system is different—from the basic combination of a consumer, processing logic, and producer to involved clusters like Spark Streaming with its machine learning libraries, and much in between. But there are some basic design patterns, which are known solutions to common requirements of stream-processing architectures. We'll review a few of those well-known patterns and show how they are used with a few examples.

Single-Event Processing

The most basic pattern of stream processing is the processing of each event in isolation. This is also known as a map/filter pattern because it is commonly used to filter unnecessary events from the stream or transform each event. (The term “map” is based on the map/reduce pattern in which the map stage transforms events and the reduce stage aggregates them.)

In this pattern, the stream-processing app consumes events from the stream, modifies each event, and then produces the events to another stream. An example is an app that reads log messages from a stream and writes ERROR events into a high-priority stream and the rest of the events into a low-priority stream. Another example is an application that reads events from a stream and modifies them from JSON to Avro. Such applications need to maintain state within the application because each event can be handled independently. This means that recovering from app failures or load-balancing is incredibly easy as there is no need to recover state; you can simply hand off the events to another instance of the app to process.

This pattern can be easily handled with a simple producer and consumer, as seen in [Figure 11-3](#).

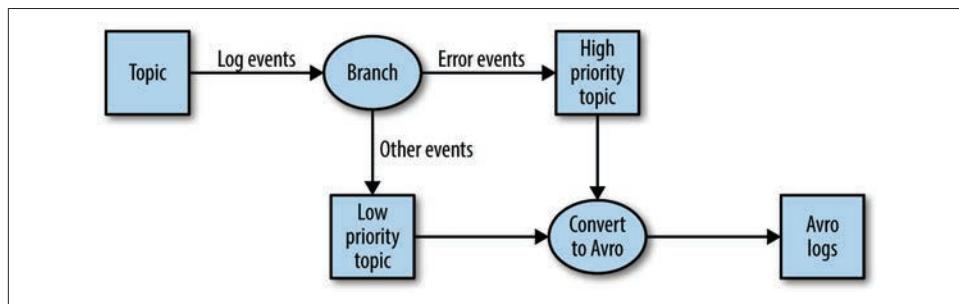


Figure 11-3. Single-event processing topology

Processing with Local State

Most stream-processing applications are concerned with aggregating information, especially time-window aggregation. An example of this is finding the minimum and maximum stock prices for each day of trading and calculating a moving average.

These aggregations require maintaining a *state* for the stream. In our example, in order to calculate the minimum and average price each day, we need to store the minimum and maximum values we've seen up until the current time and compare each new value in the stream to the stored minimum and maximum.

All these can be done using *local state* (rather than a shared state) because each operation in our example is a *group by* aggregate. That is, we perform the aggregation per stock symbol, not on the entire stock market in general. We use a Kafka partitioner to make sure that all events with the same stock symbol are written to the same partition. Then, each instance of the application will get all the events from the partitions that are assigned to it (this is a Kafka consumer guarantee). This means that each instance of the application can maintain state for the subset of stock symbols that are written to the partitions that are assigned to it. See [Figure 11-4](#).

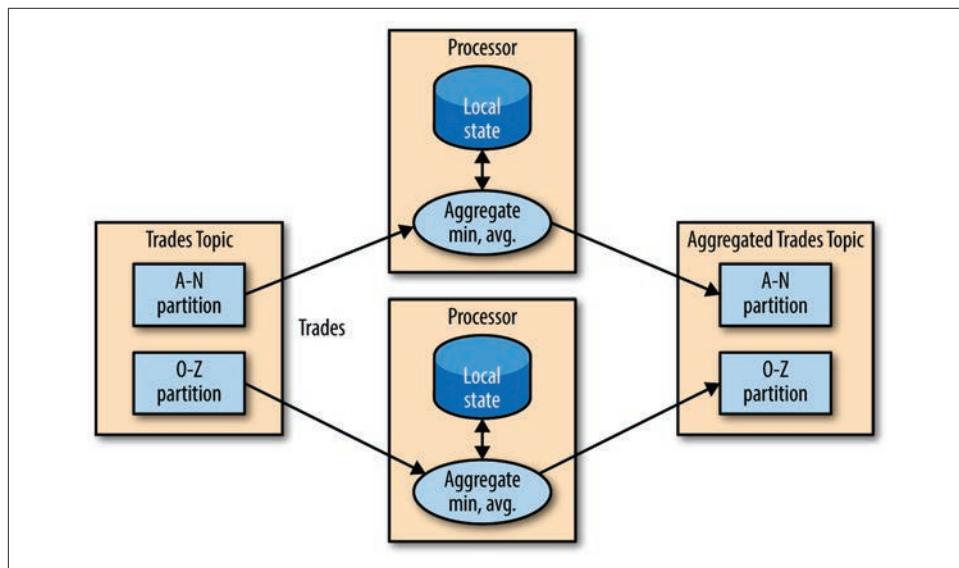


Figure 11-4. Topology for event processing with local state

Stream-processing applications become significantly more complicated when the application has local state and there are several issues a stream-processing application must address:

Memory usage

The local state must fit into the memory available to the application instance.

Persistence

We need to make sure the state is not lost when an application instance shuts down, and that the state can be recovered when the instance starts again or is replaced by a different instance. This is something that Kafka Streams handles very well—local state is stored in-memory using embedded RocksDB, which also persists the data to disk for quick recovery after restarts. But all the changes to the local state are also sent to a Kafka topic. If a stream's node goes down, the local state is not lost—it can be easily recreated by rereading the events from the Kafka topic. For example, if the local state contains “current minimum for IBM=167.19,” we store this in Kafka, so that later we can repopulate the local cache from this data. Kafka uses log compaction for these topics to make sure they don't grow endlessly and that recreating the state is always feasible.

Rebalancing

Partitions sometimes get reassigned to a different consumer. When this happens, the instance that loses the partition must store the last good state, and the instance that receives the partition must know to recover the correct state.

Stream-processing frameworks differ in how much they help the developer manage the local state they need. If your application requires maintaining local state, be sure to check the framework and its guarantees. We'll include a short comparison guide at the end of the chapter, but as we all know, software changes quickly and stream-processing frameworks doubly so.

Multiphase Processing/Repartitioning

Local state is great if you need a *group by* type of aggregate. But what if you need a result that uses all available information? For example, suppose we want to publish the top 10 stocks each day—the 10 stocks that gained the most from opening to closing during each day of trading. Obviously, nothing we do locally on each application instance is enough because all the top 10 stocks could be in partitions assigned to other instances. What we need is a two-phase approach. First, we calculate the daily gain/loss for each stock symbol. We can do this on each instance with a local state. Then we write the results to a new topic with a single partition. This partition will be read by a single application instance that can then find the top 10 stocks for the day. The second topic, which contains just the daily summary for each stock symbol, is obviously much smaller with significantly less traffic than the topics that contain the trades themselves, and therefore it can be processed by a single instance of the application. Sometimes more steps are needed to produce the result. See [Figure 11-5](#).

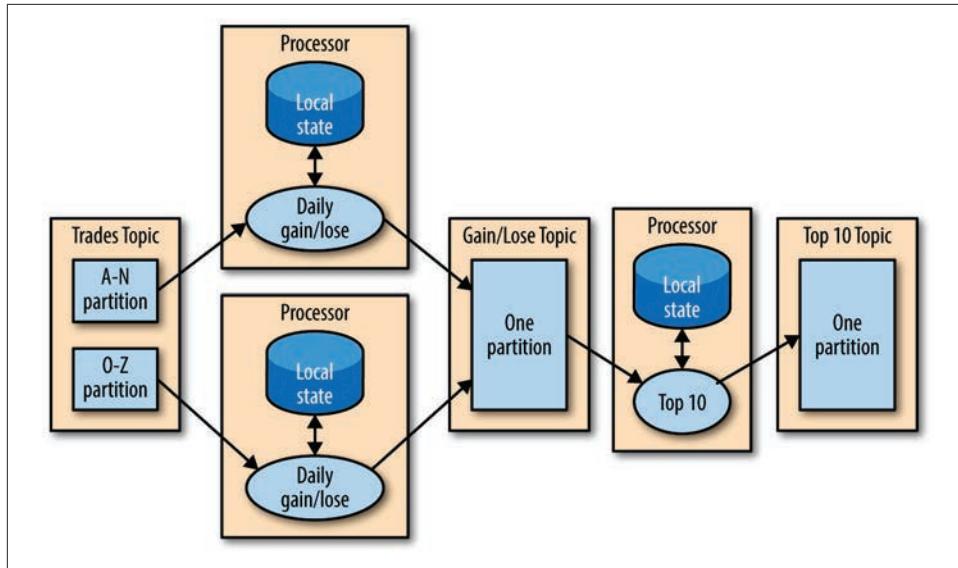


Figure 11-5. Topology that includes both local state and repartitioning steps

This type of multiphase processing is very familiar to those who write map-reduce code, where you often have to resort to multiple reduce phases. If you've ever written map-reduce code, you'll remember that you needed a separate app for each reduce step. Unlike MapReduce, most stream-processing frameworks allow including all steps in a single app, with the framework handling the details of which application instance (or worker) will run each step.

Processing with External Lookup: Stream-Table Join

Sometimes stream processing requires integration with data external to the stream—validating transactions against a set of rules stored in a database, or enriching click-stream information with data about the users who clicked.

The obvious idea on how to perform an external lookup for data enrichment is something like this: for every click event in the stream, look up the user in the profile database and write an event that includes the original click plus the user age and gender to another topic. See [Figure 11-6](#).

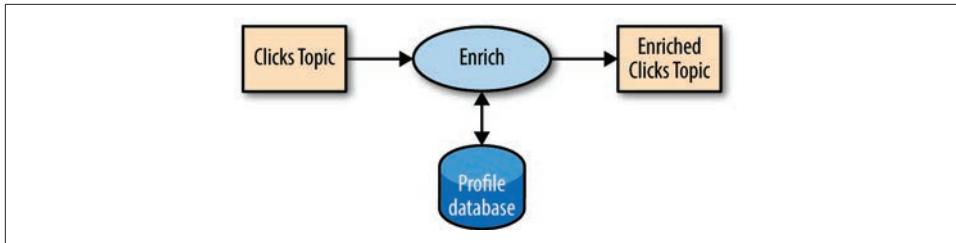


Figure 11-6. Stream processing that includes an external data source

The problem with this obvious idea is that an external lookup adds significant latency to the processing of every record—usually between 5-15 milliseconds. In many cases, this is not feasible. Often the additional load this places on the external datastore is also not acceptable—stream-processing systems can often handle 100K-500K events per second, but the database can only handle perhaps 10K events per second at reasonable performance. We want a solution that scales better.

In order to get good performance and scale, we need to cache the information from the database in our stream-processing application. Managing this cache can be challenging though—how do we prevent the information in the cache from getting stale? If we refresh events too often, we are still hammering the database and the cache isn’t helping much. If we wait too long to get new events, we are doing stream processing with stale information.

But if we can capture all the changes that happen to the database table in a stream of events, we can have our stream-processing job listen to this stream and update the cache based on database change events. Capturing changes to the database as events in a stream is known as CDC, and if you use Kafka Connect you will find multiple connectors capable of performing CDC and converting database tables to a stream of change events. This allows you to keep your own private copy of the table, and you will be notified whenever there is a database change event so you can update your own copy accordingly. See [Figure 11-7](#).

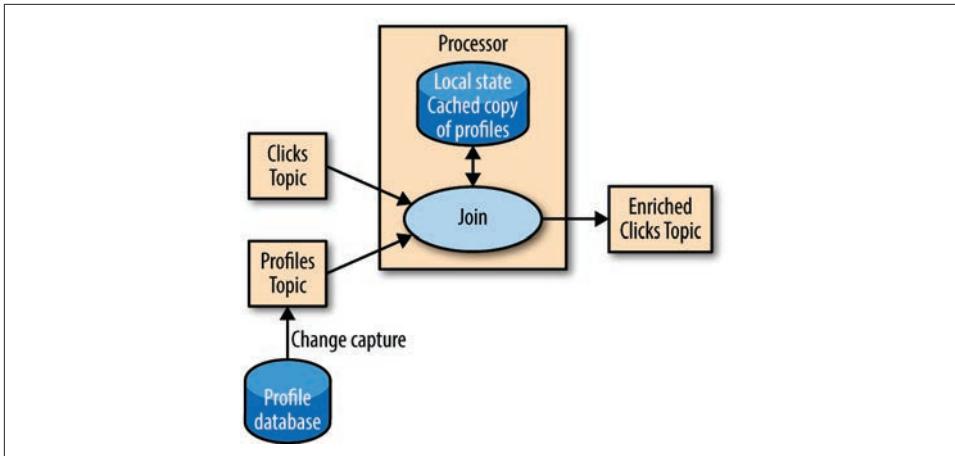


Figure 11-7. Topology joining a table and a stream of events, removing the need to involve an external data source in stream processing

Then, when you get click events, you can look up the user_id at your local cache and enrich the event. And because you are using a local cache, this scales a lot better and will not affect the database and other apps using it.

We refer to this as a *stream-table join* because one of the streams represents changes to a locally cached table.

Streaming Join

Sometimes you want to join two real event streams rather than a stream with a table. What makes a stream “real”? If you recall the discussion at the beginning of the chapter, streams are unbounded. When you use a stream to represent a table, you can ignore most of the history in the stream because you only care about the current state in the table. But when you join two streams, you are joining the entire history, trying to match events in one stream with events in the other stream that have the same key and happened in the same time-windows. This is why a streaming-join is also called a *windowed-join*.

For example, let’s say that we have one stream with search queries that people entered into our website and another stream with clicks, which include clicks on search results. We want to match search queries with the results they clicked on so that we will know which result is most popular for which query. Obviously we want to match results based on the search term but only match them within a certain time-window. We assume the result is clicked seconds after the query was entered into our search engine. So we keep a small, few-seconds-long window on each stream and match the results from each window. See Figure 11-8.

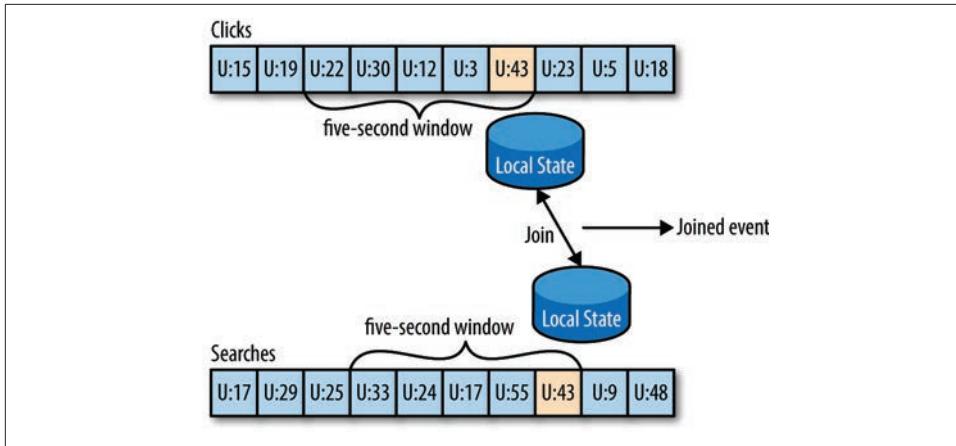


Figure 11-8. Joining two streams of events; these joins always involve a moving time window

The way this works in Kafka Streams is that both streams, queries and clicks, are partitioned on the same keys, which are also the join keys. This way, all the click events from user_id:42 end up in partition 5 of the clicks topic, and all the search events for user_id:42 end up in partition 5 of the search topic. Kafka Streams then makes sure that partition 5 of both topics is assigned to the same task. So this task sees all the relevant events for user_id:42. It maintains the join-window for both topics in its embedded RocksDB cache, and this is how it can perform the join.

Out-of-Sequence Events

Handling events that arrive at the stream at the wrong time is a challenge not just in stream processing but also in traditional ETL systems. Out-of-sequence events happen quite frequently and expectedly in IoT (Internet of Things) scenarios ([Figure 11-9](#)). For example, a mobile device loses WiFi signal for a few hours and sends a few hours' worth of events when it reconnects. This also happens when monitoring network equipment (a faulty switch doesn't send diagnostics signals until it is repaired) or manufacturing (network connectivity in plants is notoriously unreliable, especially in developing countries).

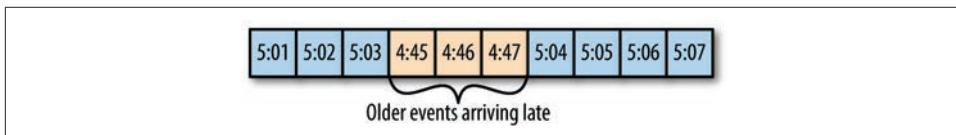


Figure 11-9. Out of sequence events

Our streams applications need to be able to handle those scenarios. This typically means the application has to do the following:

- Recognize that an event is out of sequence—this requires that the application examine the event time and discover that it is older than the current time.
- Define a time period during which it will attempt to reconcile out-of-sequence events. Perhaps a three-hour delay should be reconciled and events over three weeks old can be thrown away.
- Have an in-band capability to reconcile this event. This is the main difference between streaming apps and batch jobs. If we have a daily batch job and a few events arrived after the job completed, we can usually just rerun yesterday's job and update the events. With stream processing, there is no "rerun yesterday's job"—the same continuous process needs to handle both old and new events at any given moment.
- Be able to update results. If the results of the stream processing are written into a database, a *put* or *update* is enough to update the results. If the stream app sends results by email, updates may be trickier.

Several stream-processing frameworks, including Google's Dataflow and Kafka Streams, have built-in support for the notion of event time independent of the processing time and the ability to handle events with event times that are older or newer than the current processing time. This is typically done by maintaining multiple aggregation windows available for update in the local state and giving developers the ability to configure how long to keep those window aggregates available for updates. Of course, the longer the aggregation windows are kept available for updates, the more memory is required to maintain the local state.

The Kafka's Streams API always writes aggregation results to result topics. Those are usually **compacted topics**, which means that only the latest value for each key is preserved. In case the results of an aggregation window need to be updated as a result of a late event, Kafka Streams will simply write a new result for this aggregation window, which will overwrite the previous result.

Reprocessing

The last important pattern is processing events. There are two variants of this pattern:

- We have an improved version of our stream-processing application. We want to run the new version of the application on the same event stream as the old, produce a new stream of results that does not replace the first version, compare the results between the two versions, and at some point move clients to use the new results instead of the existing ones.
- The existing stream-processing app is buggy. We fix the bug and we want to reprocess the event stream and recalculate our results

The first use case is made simple by the fact that Apache Kafka stores the event streams in their entirety for long periods of time in a scalable datastore. This means that having two versions of a stream processing-application writing two result streams only requires the following:

- Spinning up the new version of the application as a new consumer group
- Configuring the new version to start processing from the first offset of the input topics (so it will get its own copy of all events in the input streams)
- Letting the new application continue processing and switching the client applications to the new result stream when the new version of the processing job has caught up

The second use case is more challenging—it requires “resetting” an existing app to start processing back at the beginning of the input streams, resetting the local state (so we won’t mix results from the two versions of the app), and possibly cleaning the previous output stream. While Kafka Streams has a tool for resetting the state for a stream-processing app, our recommendation is to try to use the first method whenever sufficient capacity exists to run two copies of the app and generate two result streams. The first method is much safer—it allows switching back and forth between multiple versions and comparing results between versions, and doesn’t risk losing critical data or introducing errors during the cleanup process.

Kafka Streams by Example

In order to demonstrate how these patterns are implemented in practice, we’ll show a few examples using Apache Kafka’s Streams API. We are using this specific API because it is relatively simple to use and it ships with Apache Kafka, which you already have access to. It is important to remember that the patterns can be implemented in any stream-processing framework and library—the patterns are universal but the examples are specific.

Apache Kafka has two streams APIs—a low-level Processor API and a high-level Streams DSL. We will use Kafka Streams DSL in our examples. The DSL allows you to define the stream-processing application by defining a chain of transformations to events in the streams. Transformations can be as simple as a filter or as complex as a stream-to-stream join. The lower level API allows you to create your own transformations, but as you'll see, this is rarely required.

An application that uses the DSL API always starts with using the StreamBuilder to create a processing *topology*—a directed graph (DAG) of transformations that are applied to the events in the streams. Then you create a KafkaStreams execution object from the topology. Starting the KafkaStreams object will start multiple threads, each applying the processing topology to events in the stream. The processing will conclude when you close the KafkaStreams object.

We'll look at few examples that use Kafka Streams to implement some of the design patterns we just discussed. A simple word count example will be used to demonstrate the map/filter pattern and simple aggregates. Then we'll move to an example where we calculate different statistics on stock market trades, which will allow us to demonstrate window aggregations. Finally we'll use ClickStream Enrichment as an example to demonstrate streaming joins.

Word Count

Let's walk through an abbreviated word count example for Kafka Streams. You can find the full example on [GitHub](#).

The first thing you do when creating a stream-processing app is configure Kafka Streams. Kafka Streams has a large number of possible configurations, which we won't discuss here, but you can find them in the [documentation](#). In addition, you can also configure the producer and consumer embedded in Kafka Streams by adding any producer or consumer config to the Properties object:

```
public class WordCountExample {  
  
    public static void main(String[] args) throws Exception{  
  
        Properties props = new Properties();  
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,  
                 "wordcount"); ①  
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,  
                 "localhost:9092"); ②  
        props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,  
                 Serdes.String().getClass().getName()); ③  
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,  
                 Serdes.String().getClass().getName());  
    }  
}
```

- ❶ Every Kafka Streams application must have an application ID. This is used to coordinate the instances of the application and also when naming the internal local stores and the topics related to them. This name must be unique for each Kafka Streams application working with the same Kafka cluster.
- ❷ The Kafka Streams application always reads data from Kafka topics and writes its output to Kafka topics. As we'll discuss later, Kafka Streams applications also use Kafka for coordination. So we had better tell our app where to find Kafka.
- ❸ When reading and writing data, our app will need to serialize and deserialize, so we provide default Serde classes. If needed, we can override these defaults later when building the streams topology.

Now that we have the configuration, let's build our streams topology:

```
KStreamBuilder builder = new KStreamBuilder(); ❶

KStream<String, String> source =
    builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
    Arrays.asList(pattern.split(value.toLowerCase())))
    .map((key, value) -> new KeyValue<Object,
        Object>(value, value))
    .filter((key, value) -> (!value.equals("the"))) ❸
    .groupByKey() ❹
    .count("CountStore").mapValues(value->
        Long.toString(value)).toStream(); ❺
counts.to("wordcount-output"); ❻
```

- ❶ We create a `KStreamBuilder` object and start defining a stream by pointing at the topic we'll use as our input.
- ❷ Each event we read from the source topic is a line of words; we split it up using a regular expression into a series of individual words. Then we take each word (currently a value of the event record) and put it in the event record key so it can be used in a group-by operation.
- ❸ We filter out the word "the," just to show how easy filtering is.
- ❹ And we group by key, so we now have a collection of events for each unique word.

- ⑤ We count how many events we have in each collection. The result of counting is a `Long` data type. We convert it to a `String` so it will be easier for humans to read the results.
- ⑥ Only one thing left—write the results back to Kafka.

Now that we have defined the flow of transformations that our application will run, we just need to... run it:

```
KafkaStreams streams = new KafkaStreams(builder, props); ①

streams.start(); ②

// usually the stream application would be running
// forever,
// in this example we just let it run for some time and
// stop since the input data is finite.
Thread.sleep(5000L);

streams.close(); ③

}

}
```

- ① Define a `KafkaStreams` object based on our topology and the properties we defined.
- ② Start Kafka Streams.
- ③ After a while, stop it.

Thats it! In just a few short lines, we demonstrated how easy it is to implement a single event processing pattern (we applied a map and a filter on the events). We repartitioned the data by adding a group-by operator and then maintained simple local state when we counted the number of records that have each word as a key. Then we maintained simple local state when we counted the number of times each word appeared.

At this point, we recommend running the full example. The [README in the GitHub repository](#) contains instructions on how to run the example.

One thing you'll notice is that you can run the entire example on your machine without installing anything except Apache Kafka. This is similar to the experience you may have seen when using Spark in something like *Local Mode*. The main difference is that if your input topic contains multiple partitions, you can run multiple instances of the `WordCount` application (just run the app in several different terminal tabs) and you have your first Kafka Streams processing cluster. The instances of the `WordCount` application talk to each other and coordinate the work. One of the biggest

barriers to entry with Spark is that local mode is very easy to use, but then to run a production cluster, you need to install YARN or Mesos and then install Spark on all those machines, and then learn how to submit your app to the cluster. With the Kafka's Streams API, you just start multiple instances of your app—and you have a cluster. The exact same app is running on your development machine and in production.

Stock Market Statistics

The next example is more involved—we will read a stream of stock market trading events that include the stock ticker, ask price, and ask size. In stock market trades, *ask price* is what a seller is asking for whereas *bid price* is what the buyer is suggesting to pay. *Ask size* is the number of shares the seller is willing to sell at that price. For simplicity of the example, we'll ignore bids completely. We also won't include a timestamp in our data; instead, we'll rely on event time populated by our Kafka producer.

We will then create output streams that contains a few windowed statistics:

- Best (i.e., minimum) ask price for every five-second window
- Number of trades for every five-second window
- Average ask price for every five-second window

All statistics will be updated every second.

For simplicity, we'll assume our exchange only has 10 stock tickers trading in it. The setup and configuration are very similar to those we used in the “Word Count” on page 265:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
    Constants.BROKER);
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
    TradeSerde.class.getName());
```

The main difference is the Serde classes used. In the “Word Count” on page 265, we used strings for both key and value and therefore used the `Serdes.String()` class as a serializer and deserializer for both. In this example, the key is still a string, but the value is a `Trade` object that contains the ticker symbol, ask price, and ask size. In order to serialize and deserialize this object (and a few other objects we used in this small app), we used the Gson library from Google to generate a JSON serializer and deserializer from our Java object. Then created a small wrapper that created a Serde object from those. Here is how we created the Serde:

```

static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(),
              new JsonDeserializer<Trade>(Trade.class));
    }
}

```

Nothing fancy, but you need to remember to provide a Serde object for every object you want to store in Kafka—input, output, and in some cases, also intermediate results. To make this easier, we recommend generating these Serdes through projects like GSon, Avro, Protobufs, or similar.

Now that we have everything configured, it's time to build our topology:

```

KStream<TickerWindow, TradeStats> stats = source.groupByKey() ①
    .aggregate(TradeStats::new, ②
               (k, v, tradestats) -> tradestats.add(v), ③
               TimeWindows.of(5000).advanceBy(1000), ④
               new TradeStatsSerde(), ⑤
               "trade-stats-store") ⑥
    .toStream((key, value) -> new TickerWindow(key.key(),
                                                 key.window().start())) ⑦
    .mapValues((trade) -> trade.computeAvgPrice()); ⑧

stats.to(new TickerWindowSerde(), new TradeStatsSerde(),
         "stockstats-output"); ⑨

```

- ➊ We start by reading events from the input topic and performing a `groupByKey()` operation. Despite its name, this operation does not do any grouping. Rather, it ensures that the stream of events is partitioned based on the record key. Since we wrote the data into a topic with a key and didn't modify the key before calling `groupByKey()`, the data is still partitioned by its key—so this method does nothing in this case.
- ➋ After we ensure correct partitioning, we start the windowed aggregation. The “aggregate” method will split the stream into overlapping windows (a five-second window every second), and then apply an aggregate method on all the events in the window. The first parameter this method takes is a new object that will contain the results of the aggregation—`Tradestats` in our case. This is an object we created to contain all the statistics we are interested in for each time window—minimum price, average price, and number of trades.
- ➌ We then supply a method for actually aggregating the records—in this case, an `add` method of the `Tradestats` object is used to update the minimum price, number of trades, and total prices in the window with the new record.

- ④ We define the window—in this case, a window of five seconds (5,000 ms), advancing every second.
- ⑤ Then we provide a Serde object for serializing and deserializing the results of the aggregation (the `Tradestats` object).
- ⑥ As mentioned in “[Stream-Processing Design Patterns](#)” on page 256, windowing aggregation requires maintaining a state and a local store in which the state will be maintained. The last parameter of the aggregate method is the name of the state store. This can be any unique name.
- ⑦ The results of the aggregation is a *table* with the ticker and the time window as the primary key and the aggregation result as the value. We are turning the table back into a stream of events and replacing the key that contains the entire time window definition with our own key that contains just the ticker and the start time of the window. This `toStream` method converts the table into a stream and also converts the key into my `TickerWindow` object.
- ⑧ The last step is to update the average price—right now the aggregation results include the sum of prices and number of trades. We go over these records and use the existing statistics to calculate average price so we can include it in the output stream.
- ⑨ And finally, we write the results back to the `stockstats-output` stream.

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in the “[Word Count](#)” on page 265.

This example shows how to perform windowed aggregation on a stream—probably the most popular use case of stream processing. One thing to notice is how little work was needed to maintain the local state of the aggregation—just provide a Serde and name the state store. Yet this application will scale to multiple instances and automatically recover from a failure of each instance by shifting processing of some partitions to one of the surviving instances. We will see more on how it is done in “[Kafka Streams: Architecture Overview](#)” on page 272.

As usual, you can find the complete example including instructions for running it on [GitHub](#).

Click Stream Enrichment

The last example will demonstrate streaming joins by enriching a stream of clicks on a website. We will generate a stream of simulated clicks, a stream of updates to a fictional profile database table, and a stream of web searches. We will then join all three

streams to get a 360-view into each user activity. What did the users search for? What did they click as a result? Did they change their “interests” in their user profile? These kinds of joins provide a rich data collection for analytics. Product recommendations are often based on this kind of information—user searched for bikes, clicked on links for “Trek,” and is interested in travel, so we can advertise bikes from Trek, helmets, and bike tours to exotic locations like Nebraska.

Since configuring the app is similar to the previous examples, let’s skip this part and take a look at the topology for joining multiple streams:

```
KStream<Integer, PageView> views =
builder.stream(Serdes.Integer(),
new PageViewSerde(), Constants.PAGE_VIEW_TOPIC); ①
KStream<Integer, Search> searches =
builder.stream(Serdes.Integer(), new SearchSerde(),
Constants.SEARCH_TOPIC);
KTable<Integer, UserProfile> profiles =
builder.table(Serdes.Integer(), new ProfileSerde(),
Constants.USER_PROFILE_TOPIC, "profile-store"); ②

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ③
    (page, profile) -> new UserActivity(profile.getUserID(),
    profile.getUserName(), profile.getZipcode(),
    profile.getInterests(), "", page.getPage())); ④

KStream<Integer, UserActivity> userActivityKStream =
viewsWithProfile.leftJoin(searches, ⑤
    (userActivity, search) ->
    userActivity.updateSearch(search.getSearchTerms()), ⑥
    JoinWindows.of(1000), Serdes.Integer(),
    new UserActivitySerde(), new SearchSerde()); ⑦
```

- ① First, we create a streams objects for the two streams we want to join—clicks and searches.
- ② We also define a KTable for the user profiles. A KTable is a local cache that is updated through a stream of changes.
- ③ Then we enrich the stream of clicks with user-profile information by joining the stream of events with the profile table. In a stream-table join, each event in the stream receives information from the cached copy of the profile table. We are doing a left-join, so clicks without a known user will be preserved.
- ④ This is the join method—it takes two values, one from the stream and one from the record, and returns a third value. Unlike in databases, you get to decide how to combine the two values into one result. In this case, we created one `activity` object that contains both the user details and the page viewed.

- ⑤ Next, we want to `join` the click information with searches performed by the same user. This is still a left `join`, but now we are joining two streams, not streaming to a table.
- ⑥ This is the `join` method—we simply add the search terms to all the matching page views.
- ⑦ This is the interesting part—a *stream-to-stream join* is a join with a time window. Joining all clicks and searches for each user doesn’t make much sense—we want to join each search with clicks that are related to it—that is, click that occurred a short period of time after the search. So we define a join window of one second. Clicks that happen within one second of the search are considered relevant, and the search terms will be included in the activity record that contains the click and the user profile. This will allow a full analysis of searches and their results.

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in the “[Word Count](#)” on page 265.

This example shows two different join patterns possible in stream processing. One joins a stream with a table to enrich all streaming events with information in the table. This is similar to joining a fact table with a dimension when running queries on a data warehouse. The second example joins two streams based on a time window. This operation is unique to stream processing.

As usual, you can find the complete example including instructions for running it on [GitHub](#).

Kafka Streams: Architecture Overview

The examples in the previous section demonstrated how to use the Kafka Streams API to implement a few well-known stream-processing design patterns. But to understand better how Kafka’s Streams library actually works and scales, we need to peek under the covers and understand some of the design principles behind the API.

Building a Topology

Every streams application implements and executes at least one *topology*. Topology (also called DAG, or directed acyclic graph, in other stream-processing frameworks) is a set of operations and transitions that every event moves through from input to output. [Figure 11-10](#) shows the topology in the “[Word Count](#)” on page 265.

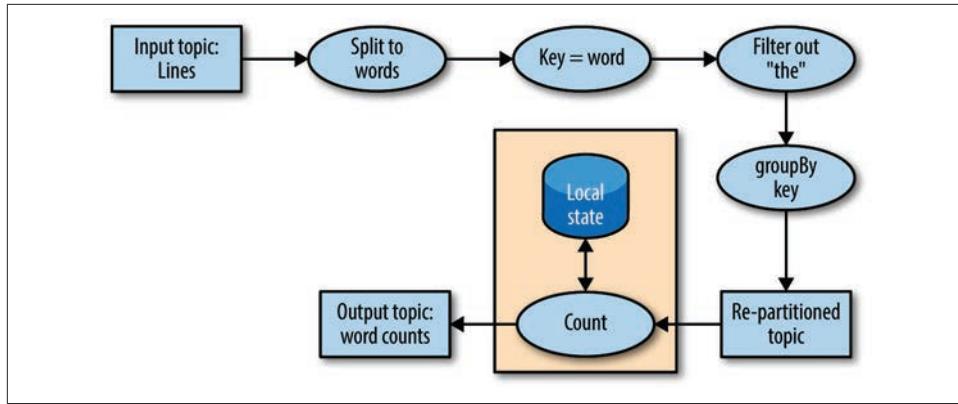


Figure 11-10. Topology for the word-count stream processing example

Even a simple app has a nontrivial topology. The topology is made up of processors—those are the nodes in the topology graph (represented by circles in our diagram). Most processors implement an operation of the data—filter, map, aggregate, etc. There are also source processors, which consume data from a topic and pass it on, and sink processors, which take data from earlier processors and produce it to a topic. A topology always starts with one or more source processors and finishes with one or more sink processors.

Scaling the Topology

Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application. You can run the Streams application on one machine with multiple threads or on multiple machines; in either case, all active threads in the application will balance the work involved in data processing.

The Streams engine parallelizes execution of a topology by splitting it into tasks. The number of tasks is determined by the Streams engine and depends on the number of partitions in the topics that the application processes. Each task is responsible for a subset of the partitions: the task will subscribe to those partitions and consume events from them. For every event it consumes, the task will execute all the processing steps that apply to this partition in order before eventually writing the result to the sink. Those tasks are the basic unit of parallelism in Kafka Streams, because each task can execute independently of others. See [Figure 11-11](#).

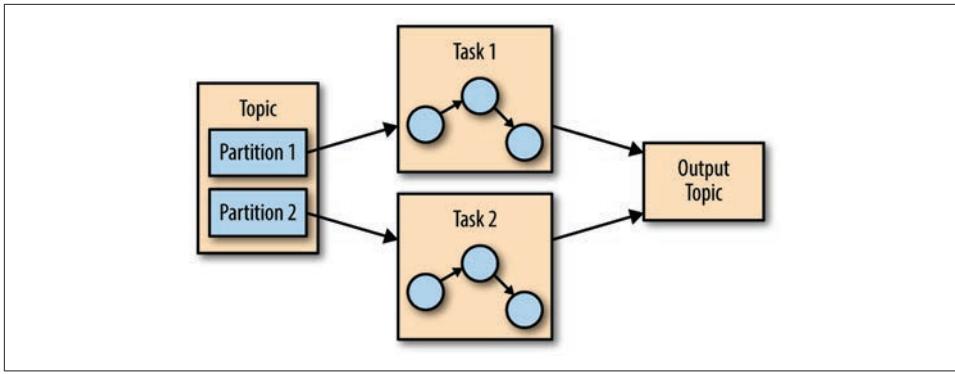


Figure 11-11. Two tasks running the same topology—one for each partition in the input topic

The developer of the application can choose the number of threads each application instance will execute. If multiple threads are available, every thread will execute a subset of the tasks that the application creates. If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server. This is the way streaming applications scale: you will have as many tasks as you have partitions in the topics you are processing. If you want to process faster, add more threads. If you run out of resources on the server, start another instance of the application on another server. Kafka will automatically coordinate work—it will assign each task its own subset of partitions and each task will independently process events from those partitions and maintain its own local state with relevant aggregates if the topology requires this. See [Figure 11-12](#).

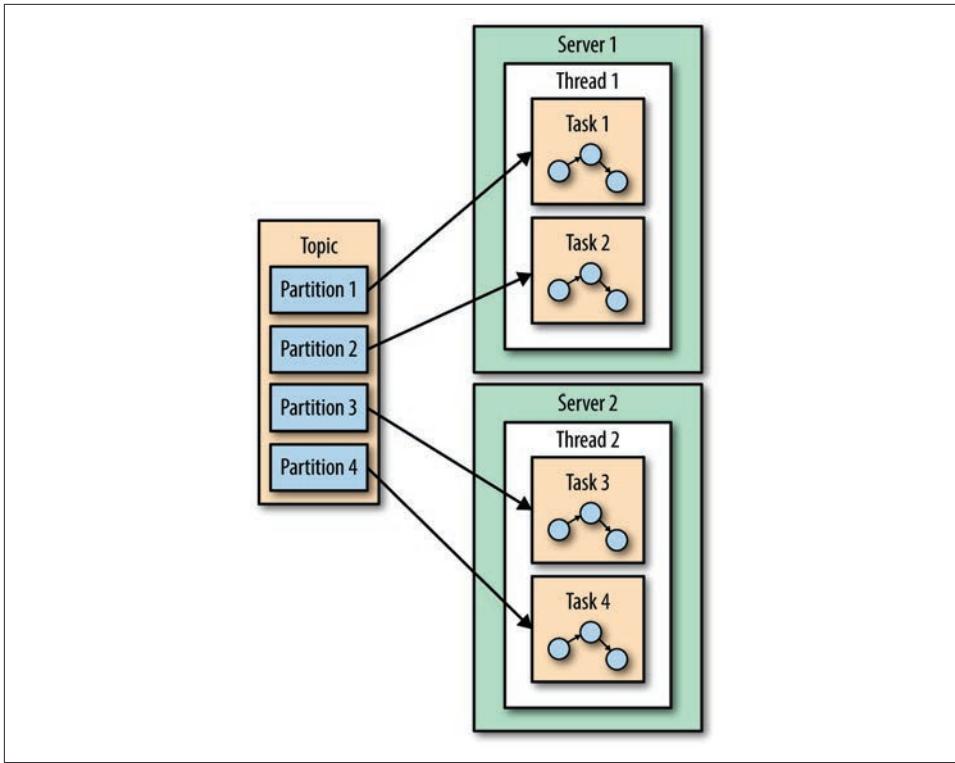


Figure 11-12. The stream processing tasks can run on multiple threads and multiple servers

You may have noticed that sometimes a processing step may require results from multiple partitions, which could create dependencies between tasks. For example, if we join two streams, as we did in the ClickStream example in “[Click Stream Enrichment](#)” on page 270, we need data from a partition in each stream before we can emit a result. Kafka Streams handles this situation by assigning all the partitions needed for one join to the same task so that the task can consume from all the relevant partitions and perform the join independently. This is why Kafka Streams currently requires that all topics that participate in a join operation will have the same number of partitions and be partitioned based on the join key.

Another example of dependencies between tasks is when our application requires repartitioning. For instance, in the ClickStream example, all our events are keyed by the user ID. But what if we want to generate statistics per page? Or per zip code? We’ll need to repartition the data by the zip code and run an aggregation of the data with the new partitions. If task 1 processes the data from partition 1 and reaches a processor that repartitions the data (groupBy operation), it will need to *shuffle*, which means sending them the events—send events to other tasks to process them. Unlike

other stream processor frameworks, Kafka Streams repartitions by writing the events to a new topic with new keys and partitions. Then another set of tasks reads events from the new topic and continues processing. The repartitioning steps break our topology into two subtopologies, each with its own tasks. The second set of tasks depends on the first, because it processes the results of the first subtopology. However, the first and second sets of tasks can still run independently and in parallel because the first set of tasks writes data into a topic at its own rate and the second set consumes from the topic and processes the events on its own. There is no communication and no shared resources between the tasks and they don't need to run on the same threads or servers. This is one of the more useful things Kafka does—reduce dependencies between different parts of a pipeline. See [Figure 11-13](#).

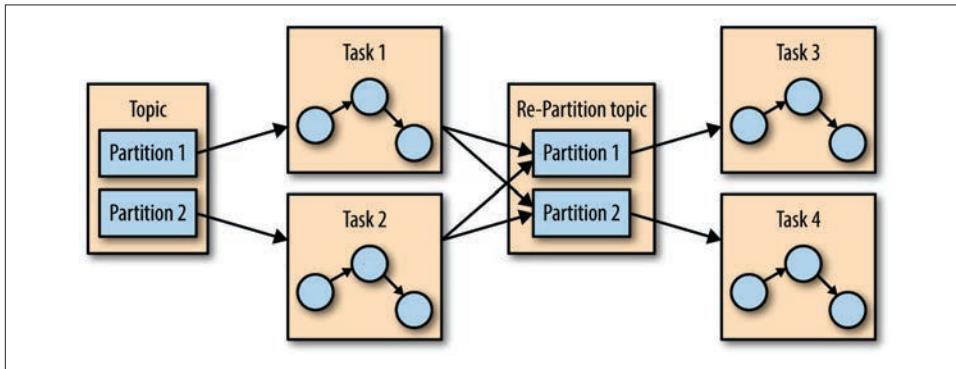


Figure 11-13. Two sets of tasks processing events with a topic for re-partitioning events between them

Surviving Failures

The same model that allows us to scale our application also allows us to gracefully handle failures. First, Kafka is highly available, and therefore the data we persist to Kafka is also highly available. So if the application fails and needs to restart, it can look up its last position in the stream from Kafka and continue its processing from the last offset it committed before failing. Note that if the local state store is lost (e.g., because we needed to replace the server it was stored on), the streams application can always re-create it from the change log it stores in Kafka.

Kafka Streams also leverages Kafka's consumer coordination to provide high availability for tasks. If a task failed but there are threads or other instances of the streams application that are active, the task will restart on one of the available threads. This is similar to how consumer groups handle the failure of one of the consumers in the group by assigning partitions to one of the remaining consumers.

Stream Processing Use Cases

Throughout this chapter we've learned how to do stream processing—from general concepts and patterns to specific examples in Kafka Streams. At this point it may be worth looking at the common stream processing use cases. As explained in the beginning of the chapter, stream processing—or continuous processing—is useful in cases where you want your events to be processed in quick order rather than wait for hours until the next batch, but also where you are not expecting a response to arrive in milliseconds. This is all true but also very abstract. Let's look at a few real scenarios that can be solved with stream processing:

Customer Service

Suppose that you just reserved a room at a large hotel chain and you expect an email confirmation and receipt. A few minutes after reserving, when the confirmation still hasn't arrived, you call customer service to confirm your reservation. Suppose the customer service desk tells you "I don't see the order in our system, but the batch job that loads the data from the reservation system to the hotels and the customer service desk only runs once a day, so please call back tomorrow. You should see the email within 2-3 business days." This doesn't sound like very good service, yet I've had this conversation more than once with a large hotel chain. What we really want is every system in the hotel chain to get an update about a new reservations seconds or minutes after the reservation is made, including the customer service center, the hotel, the system that sends email confirmations, the website, etc. You also want the customer service center to be able to immediately pull up all the details about any of your past visits to any of the hotels in the chain, and the reception desk at the hotel to know that you are a loyal customer so they can give you an upgrade. Building all those systems using stream-processing applications allows them to receive and process updates in near real time, which makes for a better customer experience. With such a system, I'd receive a confirmation email within minutes, my credit card would be charged on time, the receipt would be sent, and the service desk could immediately answer my questions regarding the reservation.

Internet of Things

Internet of Things can mean many things—from a home device for adjusting temperature and ordering refills of laundry detergent to real-time quality control of pharmaceutical manufacturing. A very common use case when applying stream processing to sensors and devices is to try to predict when preventive maintenance is needed. This is similar to application monitoring but applied to hardware and is common in many industries, including manufacturing, telecommunications (identifying faulty cellphone towers), cable TV (identifying faulty box-top devices before users complain), and many more. Every case has its own pattern, but the goal is similar: process events arriving from devices at a large

scale and identify patterns that signal that a device requires maintenance. These patterns can be dropped packets for a switch, more force required to tighten screws in manufacturing, or users restarting the box more frequently for cable TV.

- **Fraud Detection:** Also known as anomaly detection, is a very wide field that focuses on catching “cheaters” or bad actors in the system. Examples of fraud-detection applications include detecting credit card fraud, stock trading fraud, video-game cheaters, and cybersecurity risks. In all these fields, there are large benefits to catching fraud as early as possible, so a near real-time system that is capable of responding to events quickly—perhaps stopping a bad transaction before it is even approved—is much preferred to a batch job that detects fraud three days after the fact when cleanup is much more complicated. This is again a problem of identifying patterns in a large-scale stream of events.

In cyber security, there is a method known as *beaconing*. When the hacker plants malware inside the organization, it will occasionally reach outside to receive commands. It can be difficult to detect this activity since it can happen at any time and any frequency. Typically, networks are well defended against external attacks but more vulnerable to someone inside the organization reaching out. By processing the large stream of network connection events and recognizing a pattern of communication as abnormal (for example, detecting that this host typically doesn’t access those specific IPs), the security organization can be alerted early, before more harm is done.

How to Choose a Stream-Processing Framework

When choosing a stream-processing framework, it is important to consider the type of application you are planning on writing. Different types of applications call for different stream-processing solutions:

Ingest

Where the goal is to get data from one system to another, with some modification to the data on how it will make it conform to the target system.

Low milliseconds actions

Any application that requires almost immediate response. Some fraud detection use cases fall within this bucket.

Asynchronous microservices

These microservices perform a simple action on behalf of a larger business process, such as updating the inventory of a store. These applications may need to maintain a local state caching events as a way to improve performance.

Near real-time data analytics

These streaming applications perform complex aggregations and joins in order to slice and dice the data and generate interesting business-relevant insights.

The stream-processing system you will choose will depend a lot on the problem you are solving.

- If you are trying to solve an ingest problem, you should reconsider whether you want a stream processing system or a simpler ingest-focused system like Kafka Connect. If you are sure you want a stream processing system, you need to make sure it has both a good selection of connectors and high-quality connectors for the systems you are targeting.
- If you are trying to solve a problem that requires low milliseconds actions, you should also reconsider your choice of streams. Request-response patterns are often better suited to this task. If you are sure you want a stream-processing system, then you need to opt for one that supports an event-by-event low-latency model rather than one that focuses on microbatches.
- If you are building asynchronous microservices, you need a stream processing system that integrates well with your message bus of choice (Kafka, hopefully), has change capture capabilities that easily deliver upstream changes to the microservice local caches, and has the good support of a local store that can serve as a cache or materialized view of the microservice data.
- If you are building a complex analytics engine, you also need a stream-processing system with great support for a local store—this time, not for maintenance of local caches and materialized views but rather to support advanced aggregations, windows, and joins that are otherwise difficult to implement. The APIs should include support for custom aggregations, window operations, and multiple join types.

In addition to use-case specific considerations, there are a few global considerations you should take into account:

Operability of the system

Is it easy to deploy to production? Is it easy to monitor and troubleshoot? Is it easy to scale up and down when needed? Does it integrate well with your existing infrastructure? What if there is a mistake and you need to reprocess data?

Usability of APIs and ease of debugging

I've seen orders of magnitude differences in the time it takes to write a high-quality application among different versions of the same framework. Development time and time-to-market is important so you need to choose a system that makes you efficient.

Makes hard things easy

Almost every system will claim they can do advanced windowed aggregations and maintain local caches, but the question is: do they make it easy for you? Do they handle gritty details around scale and recovery, or do they supply leaky abstractions and make you handle most of the mess? The more a system exposes clean APIs and abstractions and handles the gritty details on its own, the more productive developers will be.

Community

Most stream processing applications you consider are going to be open source, and there's no replacement for a vibrant and active community. Good community means you get new and exciting features on a regular basis, the quality is relatively good (no one wants to work on bad software), bugs get fixed quickly, and user questions get answers in timely manner. It also means that if you get a strange error and Google it, you will find information about it because other people are using this system and seeing the same issues.

Summary

We started the chapter by explaining stream processing. We gave a formal definition and discussed the common attributes of the stream-processing paradigm. We also compared it to other programming paradigms.

We then discussed important stream-processing concepts. Those concepts were demonstrated with three example applications written with Kafka Streams.

After going over all the details of these example applications, we gave an overview of the Kafka Streams architecture and explained how it works under the covers. We conclude the chapter, and the book, with several examples of stream-processing use cases and advice on how to compare different stream-processing frameworks.

1

Real-Time Processing and Storm Introduction

With the exponential growth in the amount of data being generated and advanced data-capturing capabilities, enterprises are facing the challenge of making sense out of this mountain of raw data. On the batch processing front, Hadoop has emerged as the go-to framework to deal with big data. Until recently, there has been a void when one looks for frameworks to build real-time stream processing applications. Such applications have become an integral part of a lot of businesses as they enable them to respond swiftly to events and adapt to changing situations. Examples of this are monitoring social media to analyze public response to any new product that you launch and predicting the outcome of an election based on the sentiments of election-related posts.

Organizations are collecting a large volume of data from external sources and want to evaluate/process the data in real time to get market trends, detect fraud, identify user behavior, and so on. The need for real-time processing is increasing day by day and we require a real-time system/platform that should support the following features:

- **Scalable:** The platform should be horizontally scalable without any down time.
- **Fault tolerance:** The platform should be able to process the data even after some of the nodes in a cluster go down.
- **No data lost:** The platform should provide the guaranteed processing of messages.
- **High throughput:** The system should be able to support millions of records per second and also support any size of messages.

- **Easy to operate:** The system should have easy installation and operation. Also, the expansion of clusters should be an easy process.
- **Multiple languages:** The platform should support multiple languages. The end user should be able to write code in different languages. For example, a user can write code in Python, Scala, Java, and so on. Also, we can execute different language code inside the one cluster.
- **Cluster isolation:** The system should support isolation so that dedicated processes can be assigned to dedicated machines for processing.

Apache Storm

Apache Storm has emerged as the platform of choice for industry leaders to develop distributed, real-time, data processing platforms. It provides a set of primitives that can be used to develop applications that can process a very large amount of data in real time in a highly scalable manner.

Storm is to real-time processing what Hadoop is to batch processing. It is open source software, and managed by Apache Software Foundation. It has been deployed to meet real-time processing needs by companies such as Twitter, Yahoo!, and Flipboard. Storm was first developed by Nathan Marz at BackType, a company that provided social search applications. Later, BackType was acquired by Twitter, and it is a critical part of their infrastructure. Storm can be used for the following use cases:

- **Stream processing:** Storm is used to process a stream of data and update a variety of databases in real time. This processing occurs in real time and the processing speed needs to match the input data speed.
- **Continuous computation:** Storm can do continuous computation on data streams and stream the results to clients in real time. This might require processing each message as it comes in or creating small batches over a short time. An example of continuous computation is streaming trending topics on Twitter into browsers.
- **Distributed RPC:** Storm can parallelize an intense query so that you can compute it in real time.
- **Real-time analytics:** Storm can analyze and respond to data that comes from different data sources as they happen in real time.

In this chapter, we will cover the following topics:

- What is a Storm?
- Features of Storm
- Architecture and components of a Storm cluster
- Terminologies of Storm
- Programming language
- Operation modes

Features of Storm

The following are some of the features of Storm that make it a perfect solution to process streams of data in real time:

- **Fast:** Storm has been reported to process up to 1 million tuples/records per second per node.
- **Horizontally scalable:** Being fast is a necessary feature to build a high volume/velocity data processing platform, but a single node will have an upper limit on the number of events that it can process per second. A node represents a single machine in your setup that executes Storm applications. Storm, being a distributed platform, allows you to add more nodes to your Storm cluster and increase the processing capacity of your application. Also, it is linearly scalable, which means that you can double the processing capacity by doubling the nodes.
- **Fault tolerant:** Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker, and if the node on which the worker is running dies, Storm will restart that worker on some other node in the cluster. This feature will be covered in more detail in [Chapter 3, Storm Parallelism and Data Partitioning](#).
- **Guaranteed data processing:** Storm provides strong guarantees that each message entering a Storm process will be processed at least once. In the event of failures, Storm will replay the lost tuples/records. Also, it can be configured so that each message will be processed only once.
- **Easy to operate:** Storm is simple to deploy and manage. Once the cluster is deployed, it requires little maintenance.
- **Programming language agnostic:** Even though the Storm platform runs on **Java virtual machine (JVM)**, the applications that run over it can be written in any programming language that can read and write to standard input and output streams.

Storm components

A Storm cluster follows a master-slave model where the master and slave processes are coordinated through ZooKeeper. The following are the components of a Storm cluster.

Nimbus

The Nimbus node is the master in a Storm cluster. It is responsible for distributing the application code across various worker nodes, assigning tasks to different machines, monitoring tasks for any failures, and restarting them as and when required.

Nimbus is stateless and stores all of its data in ZooKeeper. There is a single Nimbus node in a Storm cluster. If the active node goes down, then the passive node will become an Active node. It is designed to be fail-fast, so when the active Nimbus dies, the passive node will become an active node, or the down node can be restarted without having any effect on the tasks already running on the worker nodes. This is unlike Hadoop, where if the JobTracker dies, all the running jobs are left in an inconsistent state and need to be executed again. The Storm workers can work smoothly even if all the Nimbus nodes go down but the user can't submit any new jobs into the cluster or the cluster will not be able to reassign the failed workers to another node.

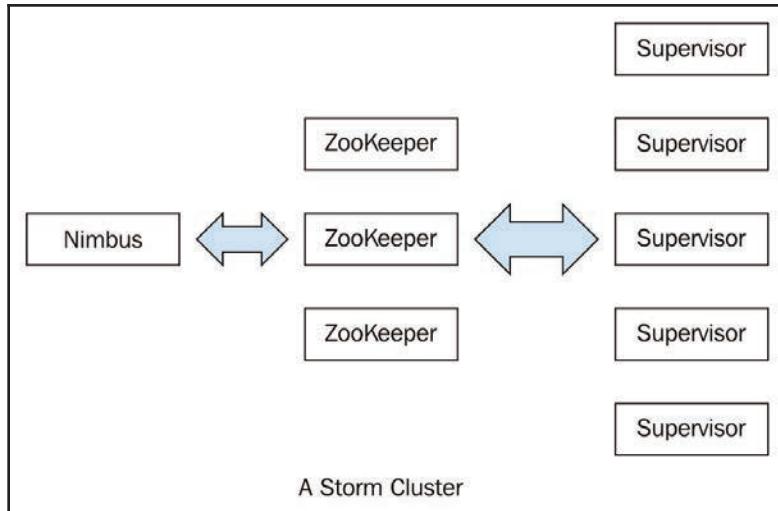
Supervisor nodes

Supervisor nodes are the worker nodes in a Storm cluster. Each supervisor node runs a supervisor daemon that is responsible for creating, starting, and stopping worker processes to execute the tasks assigned to that node. Like Nimbus, a supervisor daemon is also fail-fast and stores all of its states in ZooKeeper so that it can be restarted without any state loss. A single supervisor daemon normally handles multiple worker processes running on that machine.

The ZooKeeper cluster

In any distributed application, various processes need to coordinate with each other and share some configuration information. ZooKeeper is an application that provides all these services in a reliable manner. As a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to Storm are stored in ZooKeeper. Nimbus and supervisor nodes do not communicate directly with each other, but through ZooKeeper. As all data is stored in ZooKeeper, both Nimbus and the supervisor daemons can be killed abruptly without adversely affecting the cluster.

The following is an architecture diagram of a Storm cluster:



The Storm data model

The basic unit of data that can be processed by a Storm application is called a tuple. Each tuple consists of a predefined list of fields. The value of each field can be a byte, char, integer, long, float, double, Boolean, or byte array. Storm also provides an API to define your own datatypes, which can be serialized as fields in a tuple.

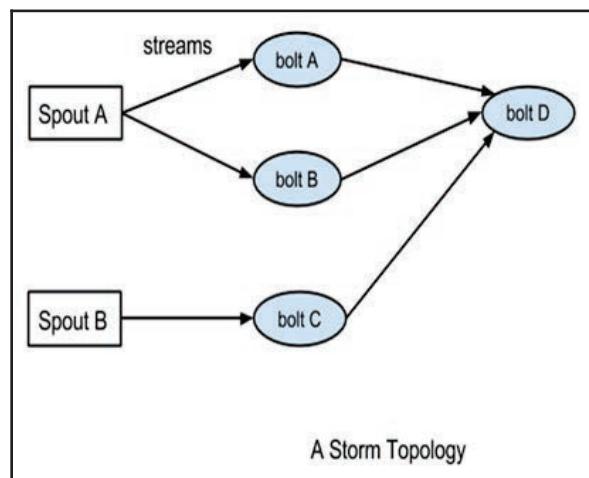
A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their datatype. The choice of dynamic typing helps to simplify the API and makes it easy to use. Also, since a processing unit in Storm can process multiple types of tuples, it's not practical to declare field types.

Each of the fields in a tuple can be accessed by its name, `getValueByField(String)`, or its positional index, `getValue(int)`, in the tuple. Tuples also provide convenient methods such as `getIntegerByField(String)` that save you from typecasting the objects. For example, if you have a *Fraction (numerator, denominator)* tuple, representing fractional numbers, then you can get the value of the numerator by either using `getIntegerByField("numerator")` or `getInteger(0)`.

You can see the full set of operations supported by `org.apache.storm.tuple.Tuple` in the Java doc that is located at
<https://storm.apache.org/releases/1.0.2/javadocs/org/apache/storm/tuple/Tuple.html>.

Definition of a Storm topology

In Storm terminology, a topology is an abstraction that defines the graph of the computation. You create a Storm topology and deploy it on a Storm cluster to process data. A topology can be represented by a direct acyclic graph, where each node does some kind of processing and forwards it to the next node(s) in the flow. The following diagram is a sample Storm topology:



The following are the components of a Storm topology:

- **Tuple:** A single message/record that flows between the different instances of a topology is called a tuple.
- **Stream:** The key abstraction in Storm is that of a stream. A stream is an unbounded sequence of tuples that can be processed in parallel by Storm. Each stream can be processed by a single or multiple types of bolts (the processing units in Storm, which are defined later in this section). Thus, Storm can also be viewed as a platform to transform streams. In the preceding diagram, streams are represented by arrows. Each stream in a Storm application is given an ID and the bolts can produce and consume tuples from these streams on the basis of their ID. Each stream also has an associated schema for the tuples that will flow through it.
- **Spout:** A spout is the source of tuples in a Storm topology. It is responsible for reading or listening to data from an external source, for example, by reading from a log file or listening for new messages in a queue and publishing them--emitting in Storm terminology into streams. A spout can emit multiple streams, each of a different schema. For example, it can read records of 10 fields from a log file and emit them as different streams of seven-fields tuples and four-fields tuples each.

The `org.apache.storm.spout.ISpout` interface is the interface used to define spouts. If you are writing your topology in Java, then you should use `org.apache.storm.topology.IRichSpout` as it declares methods to use with the `TopologyBuilder` API. Whenever a spout emits a tuple, Storm tracks all the tuples generated while processing this tuple, and when the execution of all the tuples in the graph of this source tuple is complete, it will send an acknowledgement back to the spout. This tracking happens only if a message ID was provided when emitting the tuple. If null was used as the message ID, this tracking will not happen.

A tuple processing timeout can also be defined for a topology, and if a tuple is not processed within the specified timeout, a fail message will be sent back to the spout. Again, this will happen only if you define a message ID. A small performance gain can be extracted out of Storm at the risk of some data loss by disabling the message acknowledgements, which can be done by skipping the message ID while emitting tuples.

The important methods of spout are:

- `nextTuple()`: This method is called by Storm to get the next tuple from the input source. Inside this method, you will have the logic of reading data from external sources and emitting them to an instance of `org.apache.storm.spout.ISpoutOutputCollector`. The schema for streams can be declared by using the `declareStream` method of `org.apache.storm.topology.OutputFieldsDeclarer`.

If a spout wants to emit data to more than one stream, it can declare multiple streams using the `declareStream` method and specify a stream ID while emitting the tuple. If there are no more tuples to emit at the moment, this method will not be blocked. Also, if this method does not emit a tuple, then Storm will wait for 1 millisecond before calling it again. This waiting time can be configured using the `topology.sleep.spout.wait.strategy.time.ms` setting.

- `ack (Object msgId)`: This method is invoked by Storm when the tuple with the given message ID is completely processed by the topology. At this point, the user should mark the message as processed and do the required cleaning up, such as removing the message from the message queue so that it does not get processed again.
- `fail (Object msgId)`: This method is invoked by Storm when it identifies that the tuple with the given message ID has not been processed successfully or has timed out of the configured interval. In such scenarios, the user should do the required processing so that the messages can be emitted again by the `nextTuple` method. A common way to do this is to put the message back in the incoming message queue.
- `open ()`: This method is called only once--when the spout is initialized. If it is required to connect to an external source for the input data, define the logic to connect to the external source in the `open` method, and then keep fetching the data from this external source in the `nextTuple` method to emit it further.



Another point to note while writing your spout is that none of the methods should be blocking, as Storm calls all the methods in the same thread. Every spout has an internal buffer to keep track of the status of the tuples emitted so far. The spout will keep the tuples in this buffer until they are either acknowledged or failed, calling the `ack` or `fail` method, respectively. Storm will call the `nextTuple` method only when this buffer is not full.

- **Bolt:** A bolt is the processing powerhouse of a Storm topology and is responsible for transforming a stream. Ideally, each bolt in the topology should be doing a simple transformation of the tuples, and many such bolts can coordinate with each other to exhibit a complex transformation.

The `org.apache.storm.task.IBolt` interface is preferably used to define bolts, and if a topology is written in Java, you should use the `org.apache.storm.topology.IRichBolt` interface. A bolt can subscribe to multiple streams of other components--either spouts or other bolts--in the topology and similarly can emit output to multiple streams. Output streams can be declared using the `declareStream` method of `org.apache.storm.topology.OutputFieldsDeclarer`.

The important methods of a bolt are:

- `execute(Tuple input)`: This method is executed for each tuple that comes through the subscribed input streams. In this method, you can do whatever processing is required for the tuple and then produce the output either in the form of emitting more tuples to the declared output streams, or other things such as persisting the results in a database.

You are not required to process the tuple as soon as this method is called, and the tuples can be held until required. For example, while joining two streams, when a tuple arrives you can hold it until its counterpart also comes, and then you can emit the joined tuple.

The metadata associated with the tuple can be retrieved by the various methods defined in the `Tuple` interface. If a message ID is associated with a tuple, the `execute` method must publish an `ack` or `fail` event using `OutputCollector` for the bolt, or else Storm will not know whether the tuple was processed successfully. The `org.apache.storm.topology.IBasicBolt` interface is a convenient interface that sends an acknowledgement automatically after the completion of the `execute` method. If a fail event is to be sent, this method should throw `org.apache.storm.topology.FailedException`.

- `prepare(Map stormConf, TopologyContext context, OutputCollector collector)`: A bolt can be executed by multiple workers in a Storm topology. The instance of a bolt is created on the client machine and then serialized and submitted to Nimbus. When Nimbus creates the worker instances for the topology, it sends this serialized bolt to the workers. The work will desterilize the bolt and call the `prepare` method. In this method, you should make sure the bolt is properly configured to execute tuples. Any state that you want to maintain can be stored as instance variables for the bolt that can be serialized/deserialized later.

Operation modes in Storm

Operation modes indicate how the topology is deployed in Storm. Storm supports two types of operation modes to execute the Storm topology:

- **Local mode:** In local mode, Storm topologies run on the local machine in a single JVM. This mode simulates a Storm cluster in a single JVM and is used for the testing and debugging of a topology.
- **Remote mode:** In remote mode, we will use the Storm client to submit the topology to the master along with all the necessary code required to execute the topology. Nimbus will then take care of distributing your code.

In the next chapter, we are going to cover both local and remote mode in more detail, along with a sample example.

Programming languages

Storm was designed from the ground up to be usable with any programming language. At the core of Storm is a thrift definition for defining and submitting topologies. Since thrift can be used in any language, topologies can be defined and submitted in any language.

Similarly, spouts and bolts can be defined in any language. Non-JVM spouts and bolts communicate with Storm over a JSON-based protocol over `stdin/stdout`. Adapters that implement this protocol exist for Ruby, Python, JavaScript, and Perl. You can refer to <https://github.com/apache/storm/tree/master/storm-multilang> to find out about the implementation of these adapters.

Storm-starter has an example topology, <https://github.com/apache/storm/tree/master/examples/storm-starter/multilang/resources>, which implements one of the bolts in Python.

Summary

In this chapter, we introduced you to the basics of Storm and the various components that make up a Storm cluster. We saw a definition of different deployment/operation modes in which a Storm cluster can operate.

In the next chapter, we will set up a single and three-node Storm cluster and see how we can deploy the topology on a Storm cluster. We will also see different types of stream groupings supported by Storm and the guaranteed message semantic provided by Storm.

2

Storm Deployment, Topology Development, and Topology Options

In this chapter, we are going to start with deployment of Storm on multiple node (three Storm and three ZooKeeper) clusters. This chapter is very important because it focuses on how we can set up the production Storm cluster and why we need the high availability of both the Storm Supervisor, Nimbus, and ZooKeeper (as Storm uses ZooKeeper for storing the metadata of the cluster, topology, and so on)?

The following are the key points that we are going to cover in this chapter:

- Deployment of the Storm cluster
- Program and deploy the word count example
- Different options of the Storm UI--kill, active, inactive, and rebalance
- Walkthrough of the Storm UI
- Dynamic log level settings
- Validating the Nimbus high availability

Storm prerequisites

You should have the Java JDK and ZooKeeper ensemble installed before starting the deployment of the Storm cluster.

Installing Java SDK 7

Perform the following steps to install the Java SDK 7 on your machine. You can also go with JDK 1.8:

1. Download the Java SDK 7 RPM from Oracle's site
(<http://www.oracle.com/technetwork/java/javase/downloads/index.html>).
2. Install the Java `jdk-7u<version>-linux-x64.rpm` file on your CentOS machine using the following command:

```
sudo rpm -ivh jdk-7u<version>-linux-x64.rpm
```

3. Add the following environment variable in the `~/.bashrc` file:

```
export JAVA_HOME=/usr/java/jdk<version>
```

4. Add the path of the `bin` directory of the JDK to the `PATH` system environment variable to the `~/.bashrc` file:

```
export PATH=$JAVA_HOME/bin:$PATH
```

5. Run the following command to reload the `bashrc` file on the current login terminal:

```
source ~/.bashrc
```

6. Check the Java installation as follows:

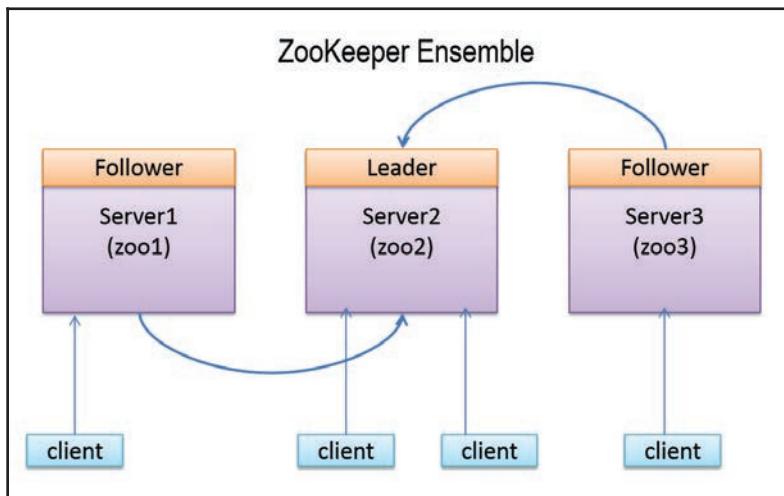
```
java -version
```

The output of the preceding command is as follows:

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b14)
Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)
```

Deployment of the ZooKeeper cluster

In any distributed application, various processes need to coordinate with each other and share configuration information. ZooKeeper is an application that provides all these services in a reliable manner. Being a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to Storm are stored in ZooKeeper. This section describes how you can set up a ZooKeeper cluster. We will be deploying a ZooKeeper ensemble of three nodes that will handle one node failure. Following is the deployment diagram of the three node ZooKeeper ensemble:



In the ZooKeeper ensemble, one node in the cluster acts as the leader, while the rest are followers. If the leader node of the ZooKeeper cluster dies, then an election for the new leader takes place among the remaining live nodes, and a new leader is elected. All write requests coming from clients are forwarded to the leader node, while the follower nodes only handle the read requests. Also, we can't increase the write performance of the ZooKeeper ensemble by increasing the number of nodes because all write operations go through the leader node.

It is advised to run an odd number of ZooKeeper nodes, as the ZooKeeper cluster keeps working as long as the majority (the number of live nodes is greater than $n/2$, where n being the number of deployed nodes) of the nodes are running. So if we have a cluster of four ZooKeeper nodes ($3 > 4/2$; only one node can die), then we can handle only one node failure, while if we had five nodes ($3 > 5/2$; two nodes can die) in the cluster, then we can handle two node failures.

Steps 1 to 4 need to be performed on each node to deploy the ZooKeeper ensemble:

1. Download the latest stable ZooKeeper release from the ZooKeeper site (<http://zookeeper.apache.org/releases.html>). At the time of writing, the latest version is ZooKeeper 3.4.6.
2. Once you have downloaded the latest version, unzip it. Now, we set up the ZK_HOME environment variable to make the setup easier.
3. Point the ZK_HOME environment variable to the unzipped directory. Create the configuration file, `zoo.cfg`, at the \$ZK_HOME/conf directory using the following commands:

```
cd $ZK_HOME/conf  
touch zoo.cfg
```

4. Add the following properties to the `zoo.cfg` file:

```
tickTime=2000  
dataDir=/var/zookeeper  
clientPort=2181  
initLimit=5  
syncLimit=2  
server.1=zoo1:2888:3888  
server.2=zoo2:2888:3888  
server.3=zoo3:2888:3888
```

Here, `zoo1`, `zoo2`, and `zoo3` are the IP addresses of the ZooKeeper nodes. The following are the definitions for each of the properties:

- `tickTime`: This is the basic unit of time in milliseconds used by ZooKeeper. It is used to send heartbeats, and the minimum session timeout will be twice the `tickTime` value.
- `dataDir`: This is the directory to store the in-memory database snapshots and transactional log.
- `clientPort`: This is the port used to listen to client connections.
- `initLimit`: This is the number of `tickTime` values needed to allow followers to connect and sync to a leader node.
- `syncLimit`: This is the number of `tickTime` values that a follower can take to sync with the leader node. If the sync does not happen within this time, the follower will be dropped from the ensemble.

The last three lines of the `server.id=host:port:port` format specify that there are three nodes in the ensemble. In an ensemble, each ZooKeeper node must have a unique ID number between 1 and 255. This ID is defined by creating a file named `myid` in the `dataDir` directory of each node. For example, the node with the ID 1 (`server.1=zoo1:2888:3888`) will have a `myid` file at directory `/var/zookeeper` with text 1 inside it.

For this cluster, create the `myid` file at three locations, shown as follows:

```
At zoo1 /var/zookeeper/myid contains 1
At zoo2 /var/zookeeper/myid contains 2
At zoo3 /var/zookeeper/myid contains 3
```

5. Run the following command on each machine to start the ZooKeeper cluster:

```
bin/zkServer.sh start
```

Check the status of the ZooKeeper nodes by performing the following steps:

6. Run the following command on the `zoo1` node to check the first node's status:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Mode: follower
```

The first node is running in `follower` mode.

7. Check the status of the second node by performing the following command:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Mode: leader
```

The second node is running in `leader` mode.

8. Check the status of the third node by performing the following command:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
```

The third node is running in `follower` mode.

9. Run the following command on the leader machine to stop the leader node:

```
bin/zkServer.sh stop
```

Now, check the status of the remaining two nodes by performing the following steps:

10. Check the status of the first node using the following command:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
```

The first node is again running in `follower` mode.

11. Check the status of the second node using the following command:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: leader
```

The third node is elected as the new leader.

12. Now, restart the third node with the following command:

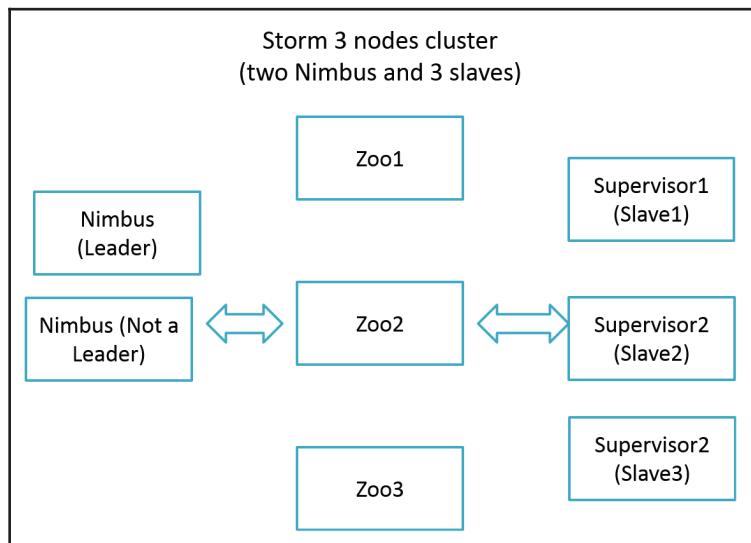
```
bin/zkServer.sh status
```

This was a quick introduction to setting up ZooKeeper that can be used for development; however, it is not suitable for production. For a complete reference on ZooKeeper administration and maintenance, please refer to the online documentation at the ZooKeeper site at <http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html>.

Setting up the Storm cluster

In this chapter, we will learn how to set up a three nodes Storm cluster, of which one node will be the active master node (Nimbus) and the other two will be worker nodes (supervisors).

The following is the deployment diagram of our three node Storm cluster:



The following are the steps that need to be performed to set up a three node Storm cluster:

1. Install and run the ZooKeeper cluster. The steps for installing ZooKeeper are mentioned in the previous section.
2. Download the latest stable Storm release from <https://storm.apache.org/downloads.html>; at the time of writing, the latest version is Storm 1.0.2.

3. Once you have downloaded the latest version, copy and unzip it in all three machines. Now, we will set the \$STORM_HOME environment variable on each machine to make the setup easier. The \$STORM_HOME environment contains the path of the Storm home folder (for example, export STORM_HOME=/home/user/storm-1.0.2).
4. Go to the \$STORM_HOME/conf directory in the master nodes and add the following lines to the storm.yaml file:

```
storm.zookeeper.servers:  
- "zoo1"  
- "zoo2"  
- "zoo3"  
storm.zookeeper.port: 2181  
nimbus.seeds: "nimbus1,nimbus2"  
storm.local.dir: "/tmp/storm-data"
```



We are installing two master nodes.

5. Go to the \$STORM_HOME/conf directory at each worker node and add the following lines to the storm.yaml file:

```
storm.zookeeper.servers:  
- "zoo1"  
- "zoo2"  
- "zoo3"  
storm.zookeeper.port: 2181  
nimbus.seeds: "nimbus1,nimbus2"  
storm.local.dir: "/tmp/storm-data"  
supervisor.slots.ports:  
- 6700  
- 6701  
- 6702  
- 6703
```



If you are planning to execute the Nimbus and supervisor on the same machine, then add the supervisor.slots.ports property to the Nimbus machine too.

6. Go to the \$STORM_HOME directory at the master nodes and execute the following command to start the master daemon:

```
$> bin/storm nimbus &
```

7. Go to the \$STORM_HOME directory at each worker node (or supervisor node) and execute the following command to start the worker daemons:

```
$> bin/storm supervisor &
```

Developing the hello world example

Before starting the development, you should have Eclipse and Maven installed in your project. The sample topology explained here will cover how to create a basic Storm project, including a spout and bolt, and how to build, and execute them.

Create a Maven project by using com.stormadvance as groupId and storm-example as artifactId.

Add the following Maven dependencies to the pom.xml file:

```
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-core</artifactId>
  <version>1.0.2</version>
  <scope>provided</scope>
</dependency>
```



Make sure the scope of the Storm dependency is provided, otherwise you will not be able to deploy the topology on the Storm cluster.

Add the following Maven build plugins in the pom.xml file:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2.1</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies
        </descriptorRef>
```

```
</descriptorRefs>
<archive>
    <manifest>
        <mainClass />
    </manifest>
</archive>
</configuration>
<executions>
    <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
            <goal>single</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>
```

Write your first sample spout by creating a `SampleSpout` class in the `com.stormadvance.storm_example` package. The `SampleSpout` class extends the serialized `BaseRichSpout` class. This spout does not connect to an external source to fetch data, but randomly generates the data and emits a continuous stream of records. The following is the source code of the `SampleSpout` class with an explanation:

```
public class SampleSpout extends BaseRichSpout {
    private static final long serialVersionUID = 1L;

    private static final Map<Integer, String> map = new HashMap<Integer, String>();
    static {
        map.put(0, "google");
        map.put(1, "facebook");
        map.put(2, "twitter");
        map.put(3, "youtube");
        map.put(4, "linkedin");
    }
    private SpoutOutputCollector spoutOutputCollector;

    public void open(Map conf, TopologyContext context, SpoutOutputCollector spoutOutputCollector) {
        // Open the spout
        this.spoutOutputCollector = spoutOutputCollector;
    }

    public void nextTuple() {
```

```
// Storm cluster repeatedly calls this method to emit a continuous
// stream of tuples.
final Random rand = new Random();
// generate the random number from 0 to 4.
int randomNumber = rand.nextInt(5);
spoutOutputCollector.emit(new Values(map.get(randomNumber)));
try{
    Thread.sleep(5000);
} catch(Exception e) {
    System.out.println("Failed to sleep the thread");
}
}

public void declareOutputFields(OutputFieldsDeclarer declarer) {

    // emit the tuple with field "site"
    declarer.declare(new Fields("site"));
}
}
```

Write your first sample bolt by creating a `SampleBolt` class within the same package. The `SampleBolt` class extends the serialized `BaseRichBolt` class. This bolt will consume the tuples emitted by the `SampleSpout` spout and will print the value of the field `site` on the console. The following is the source code of the `SampleStormBolt` class with an explanation:

```
public class SampleBolt extends BaseBasicBolt {
    private static final long serialVersionUID = 1L;

    public void execute(Tuple input, BasicOutputCollector collector) {
        // fetched the field "site" from input tuple.
        String test = input.getStringByField("site");
        // print the value of field "site" on console.
        System.out.println("##### Name of input site is : " + test);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

Create a main `SampleStormTopology` class within the same package. This class creates an instance of the spout and bolt along with the classes, and chaines them together using a `TopologyBuilder` class. This class uses `org.apache.storm.LocalCluster` to simulate the Storm cluster. The `LocalCluster` mode is used for debugging/testing the topology on a developer machine before deploying it on the Storm cluster. The following is the implementation of the main class:

```
public class SampleStormTopology {  
    public static void main(String[] args) throws AlreadyAliveException,  
    InvalidTopologyException {  
        // create an instance of TopologyBuilder class  
        TopologyBuilder builder = new TopologyBuilder();  
        // set the spout class  
        builder.setSpout("SampleSpout", new SampleSpout(), 2);  
        // set the bolt class  
        builder.setBolt("SampleBolt", new SampleBolt(),  
        4).shuffleGrouping("SampleSpout");  
        Config conf = new Config();  
        conf.setDebug(true);  
        // create an instance of LocalCluster class for  
        // executing topology in local mode.  
        LocalCluster cluster = new LocalCluster();  
        // SampleStormTopology is the name of submitted topology  
        cluster.submitTopology("SampleStormTopology", conf,  
        builder.createTopology());  
        try {  
            Thread.sleep(100000);  
        } catch (Exception exception) {  
            System.out.println("Thread interrupted exception : " + exception);  
        }  
        // kill the SampleStormTopology  
        cluster.killTopology("SampleStormTopology");  
        // shutdown the storm test cluster  
        cluster.shutdown();  
    }  
}
```

Go to your project's home directory and run the following commands to execute the topology in local mode:

```
$> cd $STORM_EXAMPLE_HOME  
$> mvn compile exec:java -Dexec.classpathScope=compile -  
Dexec.mainClass=com.stormadvance.storm_example.SampleStormTopology
```

Now create a new topology class for deploying the topology on an actual Storm cluster. Create a main `SampleStormClusterTopology` class within the same package. This class also creates an instance of the spout and bolt along with the classes, and chains them together using a `TopologyBuilder` class:

```
public class SampleStormClusterTopology {  
    public static void main(String[] args) throws AlreadyAliveException,  
    InvalidTopologyException {  
        // create an instance of TopologyBuilder class  
        TopologyBuilder builder = new TopologyBuilder();  
        // set the spout class  
        builder.setSpout("SampleSpout", new SampleSpout(), 2);  
        // set the bolt class  
        builder.setBolt("SampleBolt", new SampleBolt(),  
        4).shuffleGrouping("SampleSpout");  
        Config conf = new Config();  
        conf.setNumWorkers(3);  
        // This statement submit the topology on remote  
        // args[0] = name of topology  
        try {  
            StormSubmitter.submitTopology(args[0], conf,  
            builder.createTopology());  
        } catch (AlreadyAliveException alreadyAliveException) {  
            System.out.println(alreadyAliveException);  
        } catch (InvalidTopologyException invalidTopologyException) {  
            System.out.println(invalidTopologyException);  
        } catch (AuthorizationException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
}
```

Build your Maven project by running the following command on the projects home directory:

```
mvn clean install
```

The output of the preceding command is as follows:

```
- [INFO] -----  
- [INFO] BUILD SUCCESS  
[INFO] -----  
- [INFO] Total time: 58.326s
```

```
[INFO] Finished at:  
[INFO] Final Memory: 14M/116M  
[INFO] -----
```

We can deploy the topology to the cluster using the following Storm client command:

```
bin/storm jar jarName.jar [TopologyMainClass] [Args]
```

The preceding command runs `TopologyMainClass` with the arguments `arg1` and `arg2`. The main function of `TopologyMainClass` is to define the topology and submit it to the Nimbus machine. The `storm jar` part takes care of connecting to the Nimbus machine and uploading the JAR part.

Log in on a Storm Nimbus machine and execute the following commands:

```
$> cd $STORM_HOME  
$> bin/storm jar ~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar  
com.stormadvance.storm_example.SampleStormClusterTopology storm_example
```

In the preceding code `~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar` is the path of the `SampleStormClusterTopology` JAR that we are deploying on the Storm cluster.

The following information is displayed:

```
702 [main] INFO o.a.s.StormSubmitter - Generated ZooKeeper secret payload  
for MD5-digest: -8367952358273199959:-5050558042400210383  
793 [main] INFO o.a.s.s.a.AuthUtils - Got AutoCreds []  
856 [main] INFO o.a.s.StormSubmitter - Uploading topology jar  
/home/USER/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar to  
assigned location: /tmp/storm-data/nimbus/inbox/stormjar-d3007821-  
f87d-48af-8364-cff7abf8652d.jar  
867 [main] INFO o.a.s.StormSubmitter - Successfully uploaded topology jar  
to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-d3007821-  
f87d-48af-8364-cff7abf8652d.jar  
868 [main] INFO o.a.s.StormSubmitter - Submitting topology storm_example  
in distributed mode with conf  
{"storm.zookeeper.topology.auth.scheme": "digest", "storm.zookeeper.topology.  
auth.payload": "-8367952358273199959:-5050558042400210383", "topology.workers":  
":3"}  
1007 [main] INFO o.a.s.StormSubmitter - Finished submitting topology:  
storm_example
```

Run the `jps` command to see the number of running JVM processes as follows:

```
jps
```

The preceding command's output is:

```
26827 worker
26530 supervisor
26824 worker
26468 nimbus
26822 worker
```

In the preceding code, a `worker` is the JVM launched for the `SampleStormClusterTopology` topology.

The different options of the Storm topology

This section covers the following operations that a user can perform on the Storm cluster:

- Deactivate
- Activate
- Rebalance
- Kill
- Dynamic log level settings

Deactivate

Storm supports the deactivating a topology. In the deactivated state, spouts will not emit any new tuples into the pipeline, but the processing of the already emitted tuples will continue. The following is the command to deactivate the running topology:

```
$> bin/storm deactivate topologyName
```

Deactivate `SampleStormClusterTopology` using the following command:

```
bin/storm deactivate SampleStormClusterTopology
```

The following information is displayed:

```
0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at
localhost:6627
76 [main] INFO backtype.storm.command.deactivate - Deactivated topology:
SampleStormClusterTopology
```

Activate

Storm also supports activating a topology. When a topology is activated, spouts will again start emitting tuples. The following is the command to activate the topology:

```
$> bin/storm activate topologyName
```

Activate SampleStormClusterTopology using the following command:

```
bin/storm activate SampleStormClusterTopology
```

The following information is displayed:

```
0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at
localhost:6627
65 [main] INFO backtype.storm.command.activate - Activated topology:
SampleStormClusterTopology
```

Rebalance

The process of updating the topology parallelism at the runtime is called a **rebalance**. A more detailed information of this operation can be in Chapter 3, *Storm Parallelism and Data Partitioning*.

Kill

Storm topologies are never-ending processes. To stop a topology, we need to kill it. When killed, the topology first enters into the deactivation state, processes all the tuples already emitted into it, and then stops. Run the following command to kill SampleStormClusterTopology:

```
$> bin/storm kill SampleStormClusterTopology
```

The following information is displayed:

```
0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at
localhost:6627
80 [main] INFO backtype.storm.command.kill-topology - Killed topology:
SampleStormClusterTopology
```

Now, run the `jps` command again to see the remaining JVM processes as follows:

```
jps
```

The preceding command's output is:

```
26530 supervisor
27193 Jps
26468 nimbus
```

Dynamic log level settings

This allows the user to change the log level of topology on runtime without stopping the topology. The detailed information of this operation can be found at the end of this chapter.

Walkthrough of the Storm UI

This section will show you how we can start the Storm UI daemon. However, before starting the Storm UI daemon, we assume that you have a running Storm cluster. The Storm cluster deployment steps are mentioned in the previous sections of this chapter. Now, go to the Storm home directory (`cd $STORM_HOME`) at the leader Nimbus machine and run the following command to start the Storm UI daemon:

```
$> cd $STORM_HOME
$> bin/storm ui &
```

By default, the Storm UI starts on the 8080 port of the machine where it is started. Now, we will browse to the `http://nimbus-node:8080` page to view the Storm UI, where Nimbus node is the IP address or hostname of the the Nimbus machine.

The following is a screenshot of the Storm home page:

The screenshot shows the Storm UI interface with four main sections:

- Cluster Summary:** A table showing cluster statistics:

Version	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
1.0.2	3	0	12	12	0	0
- Nimbus Summary:** A table showing Nimbus node details:

Host	Port	Status	Version	UpTime
Nimbus1	6627	Leader	1.0.2	2h 1m 37s
Nimbus2	6627	Not a Leader	1.0.2	2h 1m 10s

Showing 1 to 2 of 2 entries
- Topology Summary:** A table showing topology details (empty):

Name	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
No data available in table									

Showing 0 to 0 of 0 entries
- Supervisor Summary:** A table showing supervisor details:

Host	Id	Uptime	Slots	Used slots	Used Mem (MB)	Version
Supervisor1	f3ea6a62-9b63-453b-b4a4-54bb681c9bf9	2h 0m 56s	4	0	0	1.0.2
Supervisor2	381fcea3-28a5-402c-b34a-c65fd0077c52	2h 1m 31s	4	0	0	1.0.2
Supervisor3	14777ac7-6959-4b21-b7f3-6db9055bf854	2h 1m 1s	4	0	0	1.0.2

Cluster Summary section

This portion of the Storm UI shows the version of Storm deployed in the cluster, the uptime of the Nimbus nodes, number of free worker slots, number of used worker slots, and so on. While submitting a topology to the cluster, the user first needs to make sure that the value of the **Free slots** column should not be zero; otherwise, the topology doesn't get any worker for processing and will wait in the queue until a workers becomes free.

Nimbus Summary section

This portion of the Storm UI shows the number of Nimbus processes that are running in a Storm cluster. The section also shows the status of the Nimbus nodes. A node with the status `Leader` is an active master while the node with the status `Not a Leader` is a passive master.

Supervisor Summary section

This portion of the Storm UI shows the list of supervisor nodes running in the cluster, along with their **Id**, **Host**, **Uptime**, **Slots**, and **Used slots** columns.

Nimbus Configuration section

This portion of the Storm UI shows the configuration of the Nimbus node. Some of the important properties are:

- supervisor.slots.ports
- storm.zookeeper.port
- storm.zookeeper.servers
- storm.zookeeper.retry.interval
- worker.childopts
- supervisor.childopts

The following is a screenshot of **Nimbus Configuration**:

Showing 1 to 3 of 3 entries	
Nimbus Configuration	
Key	Value
backpressure disruptor high watermark	0.6
backpressure disruptor low watermark	0.4
client.blobstore.class	"org.apache.storm.blobstore.NimbusBlobStore"
dev.zookeeper.path	"/tmp/dev-storm-zookeeper"
drpc.authorizer.acl.filename	"drpc-auth-acl.yaml"
drpc.authorizer.acl.strict	false
drpc.childopts	"-Xmx768m"
drpc.http.creds.plugin	"org.apache.storm.security.auth.DefaultHttpCredentialsPlugin"
drpc.http.port	3774
drpc.https.keystore.password	**
drpc.https.keystore.type	"JKS"
drpc.https.port	41
drpc.invocations.port	3773
drpc.invocations.threads	64
drpc.max_buffer_size	1048576
drpc.port	3772
drpc.queue.size	128
drpc.request.timeout.secs	600
drpc.worker.threads	64
java.library.path	"/usr/local/lib:/opt/local/lib:/usr/lib"

Topology Summary section

This portion of the Storm UI shows the list of topologies running in the Storm cluster, along with their ID, the number of workers assigned to the topology, the number of executors, number of tasks, uptime, and so on.

Let's deploy the sample topology (if it is not running already) in a remote Storm cluster by running the following command:

```
$> cd $STORM_HOME  
$> bin/storm jar ~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar  
com.stormadvance.storm_example.SampleStormClusterTopology storm_example
```

We have created the `SampleStormClusterTopology` topology by defining three worker processes, two executors for `SampleSpout`, and four executors for `SampleBolt`.

After submitting `SampleStormClusterTopology` on the Storm cluster, the user has to refresh the Storm home page.

The following screenshot shows that the row is added for `SampleStormClusterTopology` in the **Topology Summary** section. The topology section contains the name of the topology, unique ID of the topology, status of the topology, uptime, number of workers assigned to the topology, and so on. The possible values of the **Status** fields are ACTIVE, KILLED, and INACTIVE.

The screenshot displays the Storm UI interface with four main sections:

- Cluster Summary:** Shows system-level metrics: Version 1.0.2, Supervisors 3, Used slots 3, Free slots 9, Total slots 12, Executors 9, and Tasks 9.
- Nimbus Summary:** Lists two Nimbus instances: `Nimbus1` (Leader, Port 6627, Version 1.0.2, UpTime 2h 15m 32s) and `Nimbus2` (Not a Leader, Port 6627, Version 1.0.2, UpTime 2h 15m 5s).
- Topology Summary:** Shows the topology `storm_example` with details: Owner User, Status ACTIVE, Uptime 30s, Num workers 3, Num executors 9, Num tasks 9, Replication count 2, Assigned Mem (MB) 2496, and Scheduler Info.
- Supervisor Summary:** Lists three Supervisor instances: `Supervisor1`, `Supervisor2`, and `Supervisor3`, each with its ID, Uptime, Slots, Used slots, Used Mem (MB), and Version.

Storm Deployment, Topology Development, and Topology Options

Let's click on `SampleStormClusterTopology` to view its detailed statistics. There are two screenshots for this. The first one contains the information about the number of workers, executors, and tasks assigned to the `SampleStormClusterTopology` topology:

The screenshot shows the Storm UI interface with the title "Topology summary". It displays a table with the following data:

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
storm_example	storm_example-3-1484535083	User	ACTIVE	11m 17s	3	9	9	2	2496	

The next screenshot contains information about the spouts and bolts, including the number of executors and tasks assigned to each spout and bolt:

The screenshot shows the Storm UI interface with several sections:

- Topology actions:** Buttons for Activate, Deactivate, Rebalance, Kill, Debug, Stop Debug, and Change Log Level.
- Topology stats:** A table showing windowed metrics for emitted, transferred, and complete latency (ms) over 10m, 3h, 1d, and all time.
- Spouts (All time):** A table listing spouts with their executors, tasks, emitted, transferred, and latency metrics. One entry is shown: SampleSpout (2 executors, 2 tasks, 120 emitted, 120 transferred, 0.000 ms latency).
- Bolts (All time):** A table listing bolts with their executors, tasks, emitted, transferred, and latency metrics. One entry is shown: SampleBolt (4 executors, 4 tasks, 0 emitted, 0 transferred, 1.000 ms execute latency).
- Topology Visualization:** A button labeled "Show Visualization".
- Topology Configuration:** A table showing configuration settings for the topology.

The information shown in the previous screenshots is as follows:

- **Topology stats:** This section will give information about the number of tuples emitted, transferred, and acknowledged, the capacity latency, and so on, within the windows of 10 minutes, 3 hours, 1 day, and since the start of the topology
- **Spouts (All time):** This section shows the statistics of all the spouts running inside the topology
- **Bolts (All time):** This section shows the statistics of all the bolts running inside the topology
- **Topology actions:** This section allows us to perform activate, deactivate, rebalance, kill, and other operations on the topologies directly through the Storm UI:
 - **Deactivate:** Click on **Deactivate** to deactivate the topology. Once the topology is deactivated, the spout stops emitting tuples and the status of the topology changes to **INACTIVE** on the Storm UI.

The screenshot shows the Storm UI interface. At the top, there is a search bar with the placeholder "Search storm_example-3-1484535083" and a "Search" button. Below the search bar is a "Topology summary" section with a table. The table has columns: Name, Id, Owner, Status, Uptime, Num workers, Num executors, Num tasks, Replication count, Assigned Mem (MB), and Scheduler Info. A single row is present: "storm_example" with "storm_example-3-1484535083" in the Id column, "User" in the Owner column, and "INACTIVE" in the Status column (which is highlighted with a red box). The "Uptime" is listed as "13m 46s". The "Num workers" is 3, "Num executors" is 9, "Num tasks" is 9, "Replication count" is 2, and "Assigned Mem (MB)" is 2496. Below the summary is a "Topology actions" section with several buttons: Activate, Deactivate, Rebalance, Kill, Debug, Stop Debug, and Change Log Level.

Deactivating the topology does not free the Storm resource.



TIP

- **Activate:** Click on the **Activate** button to activate the topology. Once the topology is activated, the spout again starts emitting tuples.
- **Kill:** Click on the **Kill** button to destroy/kill the topology. Once the topology is killed, it will free all the Storm resources allotted to this topology. While killing the topology, the Storm will first deactivate the spouts and wait for the kill time mentioned on the alerts box so that the bolts have a chance to finish the processing of the tuples emitted by the spouts before the kill command. The following screenshot shows how we can kill the topology through the Storm UI:

The screenshot shows the Storm UI's topology actions page. At the top, there are buttons for Activate, Deactivate, Rebalance, Kill, Debug, Stop Debug, and Change Log Level. The Kill button is highlighted. Below it, the Topology stats section displays windowed metrics for emitted and transferred data over various time intervals. A modal dialog box is open, asking if the user really wants to kill the topology 'storm_example'. The dialog includes fields for specifying a wait time in seconds (set to 0) and a checkbox to prevent additional dialogs. The Spouts (All time) and Bolts (All time) sections show detailed tables of their respective components.

Let's go to the Storm UI's home page to check the status of `SampleStormClusterTopology`, as shown in the following screenshot:

The screenshot shows the Storm UI's topology summary page. It lists the topology 'storm_example' with its details: Owner (User), Status (KILLED, highlighted with a red box), Uptime (26m 15s), Num workers (3), Num executors (9), Num tasks (9), Replication count (1), Assigned Mem (MB) (2406), and Scheduler Info. A search bar is visible at the top right.

Dynamic log level settings

The dynamic log level allows us to change the log level setting of the topology on the runtime from the Storm CLI and the Storm UI.

Updating the log level from the Storm UI

Go through the following steps to update the log level from the Storm UI:

1. Deploy `SampleStormClusterTopology` again on the Storm cluster if it is not running.
2. Browse the Storm UI at `http://nimbus-node:8080/`.
3. Click on the `storm_example` topology.

- Now click on the **Change Log Level** button to change the ROOT logger of the topology, as shown in the following are the screenshots:

Topology actions

Activate Deactivate Rebalance Kill Debug Stop Debug Change Log Level

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
com.your.organization.Log	Pick Level	30		Add

- Configure the entries mentioned in the following screenshots change the ROOT logger to **ERROR**:

Topology actions

Activate Deactivate Rebalance Kill Debug Stop Debug Change Log Level

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
ROOT	ERROR	0		Apply Clear Add
com.your.organization.Log	Pick Level	30		

- If you are planning to change the logging level to **DEBUG**, then you must specify the timeout (expiry time) for that log level, as shown in the following screenshots:

Topology actions

Activate Deactivate Rebalance Kill Debug Stop Debug Change Log Level

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
ROOT	DEBUG	0		Add

Topology stats

Window	Emitted	Transferred
10m 0s	160	160
3h 0m 0s	2860	2860
1d 0h 0m 0s	2860	2860
All time	2860	2860

10.191.209.12:8080 says:
You must provide a timeout > 0 for DEBUG log level. What timeout would you like (secs)?

 Prevent this page from creating additional dialogs.
OK Cancel

7. Once the time mentioned in the expiry time is reached, the log level will go back to the default value:

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler info
storm_example	storm_example-4-1484537848	ajain	ACTIVE	2h 4m 51s	3	9	9	1	2496	

8. **Clear** button mentioned in the **Action** column will clear the log setting, and the application will set the default log setting again.

Updating the log level from the Storm CLI

We can modify the log level from the Storm CLI. The following is the command that the user has to execute from the Storm directory to update the log settings on the runtime:

```
bin/storm set_log_level [topology name] -l [logger name]=[LEVEL]:[TIMEOUT]
```

In the preceding code, `topology name` is the name of the topology, and `logger name` is the logger we want to change. If you want to change the `ROOT` logger, then use `ROOT` as a value of `logger name`. The `LEVEL` is the log level you want to apply. The possible values are `DEBUG`, `INFO`, `ERROR`, `TRACE`, `ALL`, `WARN`, `FATAL`, and `OFF`.

The `TIMEOUT` is the time in seconds. The log level will go back to normal after the timeout time. The value of `TIMEOUT` is mandatory if you are setting the log level to `DEBUG/ALL`.

The following is the command to change the log level setting for the `storm_example` topology:

```
$> bin/storm set_log_level storm_example -l ROOT=DEBUG:30
```

The following is the command to clear the log level setting:

```
$> ./bin/storm set_log_level storm_example -r ROOT
```

Summary

In this chapter, we have covered the installation of Storm and ZooKeeper clusters, the deployment of topologies on Storm clusters, the high availability of Nimbus nodes, and topology monitoring through the Storm UI. We have also covered the different operations a user can perform on running topology. Finally, we focused on how we can change the log level of running topology.

In the next chapter, we will focus on the distribution of topologies on multiple Storm machines/nodes.

3

Storm Parallelism and Data Partitioning

In the first two chapters, we have covered the introduction to Storm, the installation of Storm, and developing a sample topology. In this chapter, we are focusing on distribution of the topology on multiple Storm machines/nodes. This chapter covers the following points:

- Parallelism of topology
- How to configure parallelism at the code level
- Different types of stream groupings in a Storm cluster
- Guaranteed message processing
- Tick tuple

Parallelism of a topology

Parallelism means the distribution of jobs on multiple nodes/instances where each instance can work independently and can contribute to the processing of data. Let's first look at the processes/components that are responsible for the parallelism of a Storm cluster.

Worker process

A Storm topology is executed across multiple supervisor nodes in the Storm cluster. Each of the nodes in the cluster can run one or more JVMs called **worker processes**, which are responsible for processing a part of the topology.

A worker process is specific to one of the specific topologies and can execute multiple components of that topology. If multiple topologies are being run at the same time, none of them will share any of the workers, thus providing some degree of isolation between topologies.

Executor

Within each worker process, there can be multiple threads executing parts of the topology. Each of these threads is called an **executor**. An executor can execute only one of the components, that is, any spout or bolt in the topology.

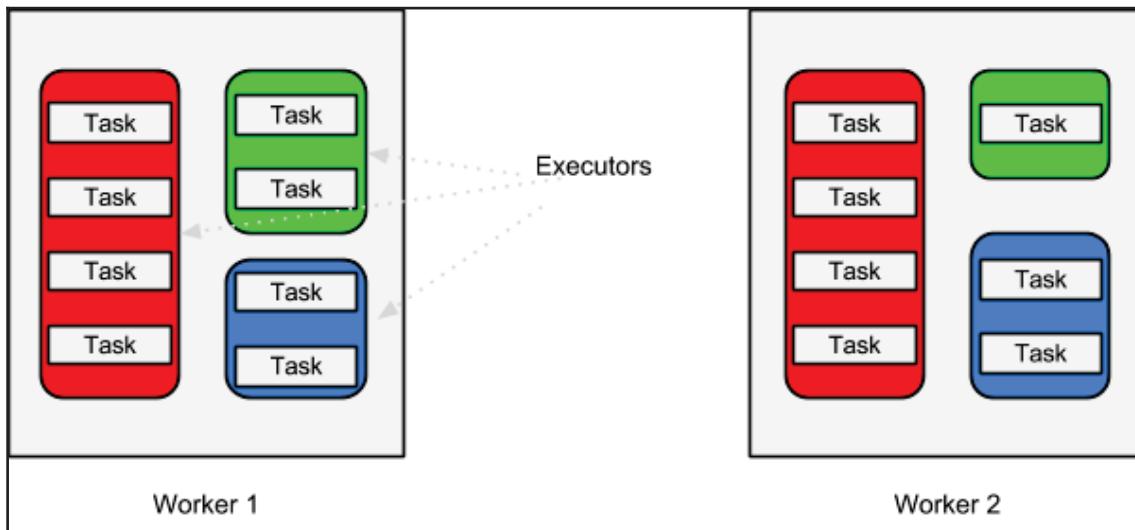
Each executor, being a single thread, can execute only tasks assigned to it serially. The number of executors defined for a spout or bolt can be changed dynamically while the topology is running, which means that you can easily control the degree of parallelism of various components in your topology.

Task

This is the most granular unit of task execution in Storm. Each task is an instance of a spout or bolt. When defining a Storm topology, you can specify the number of tasks for each spout and bolt. Once defined, the number of tasks cannot be changed for a component at runtime. Each task can be executed alone or with another task of the same type, or another instance of the same spout or bolt.

The following diagram depicts the relationship between a worker process, executors, and tasks. In the following diagram, there are two executors for each component, with each hosting a different number of tasks.

Also, as you can see, there are two executors and eight tasks defined for one component (each executor is hosting four tasks). If you are not getting enough performance out of this configuration, you can easily change the number of executors for the component to four or eight to increase performance and the tasks will be uniformly distributed between all executors of that component. The following diagrams show the relationship between executor, task, and worker:



Configure parallelism at the code level

Storm provides an API to set the number of worker processes, number of executors, and number of tasks at the code level. The following section shows how we can configure parallelism at the code level.

We can set the number of worker processes at the code level by using the `setNumWorkers` method of the `org.apache.storm.Config` class. Here is the code snippet to show these settings in practice:

```
Config conf = new Config();
conf.setNumWorkers(3);
```

In the previous chapter, we configured the number of workers as three. Storm will assign the three workers for the `SampleStormTopology` and `SampleStormClusterTopology` topology.

We can set the number of executors at the code level by passing the `parallelism_hint` argument in the `setSpout(args, args, parallelism_hint)` or `setBolt(args, args, parallelism_hint)` methods of the `org.apache.storm.topology.TopologyBuilder` class. Here is the code snippet to show these settings in practice:

```
builder.setSpout("SampleSpout", new SampleSpout(), 2);
// set the bolt class
builder.setBolt("SampleBolt", new SampleBolt(),
4).shuffleGrouping("SampleSpout");
```

In the previous chapter, we set `parallelism_hint=2` for `SampleSpout` and `parallelism_hint=4` for `SampleBolt`. At the time of execution, Storm will assign two executors for `SampleSpout` and four executors for `SampleBolt`.

We can configure the number of tasks that can execute inside the executors. Here is the code snippet to show these settings in practice:

```
builder.setSpout("SampleSpout", new SampleSpout(), 2).setNumTasks(4);
```

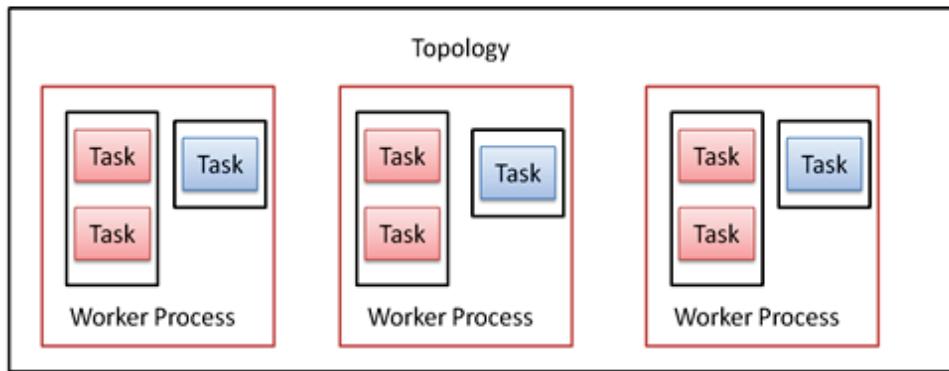
In the preceding code, we have configured the two executors and four tasks of `SampleSpout`. For `SampleSpout`, Storm will assign two tasks per executor. By default, Storm will run one task per executor if the user does not set the number of tasks at the code level.

Worker process, executor, and task distribution

Let's assume the numbers of worker processes set for the topology is three, the number of executors for `SampleSpout` is three, and the number of executors for `SampleBolt` is three. Also, the number of tasks for `SampleBolt` is to be six, meaning that each `SampleBolt` executor will have two tasks. The following diagram shows what the topology would look like in operation:

Number of worker processes =3
Number of executors for spout = 3.
Number of tasks on each spout executor = 1.
Number of executor for bolt =3.
Number of tasks on each bolt executor =2.

Total parallelism = (Number of spout tasks) +
(Number of bolt tasks)
 $=> 3 + 3*2 => 9$



Rebalance the parallelism of a topology

As explained in the previous chapter, one of the key features of Storm is that it allows us to modify the parallelism of a topology at runtime. The process of updating a topology's parallelism at runtime is called **rebalance**.

There are two ways to rebalance the topology:

- Using Storm Web UI
- Using Storm CLI

The Storm Web UI was covered in the previous chapter. This section covers how we can rebalance the topology using the Storm CLI tool. Here are the commands that we need to execute on Storm CLI to rebalance the topology:

```
> bin/storm rebalance [TopologyName] -n [NumberOfWorkers] -e  
[Spout]=[NumberOfExecutors] -e [Bolt1]=[NumberOfExecutors]  
[Bolt2]=[NumberOfExecutors]
```

The rebalance command will first deactivate the topology for the duration of the message timeout and then redistribute the workers evenly around the Storm cluster. After a few seconds or minutes, the topology will revert to the previous state of activation and restart the processing of input streams.

Rebalance the parallelism of a SampleStormClusterTopology topology

Let's first check the numbers of worker processes that are running in the Storm cluster by running the `jps` command on the supervisor machine:

Run the `jps` command on supervisor-1:

```
> jps  
24347 worker  
23940 supervisor  
24593 Jps  
24349 worker
```

Two worker processes are assigned to the supervisor-1 machine.

Now, run the `jps` command on supervisor-2:

```
> jps  
24344 worker  
23941 supervisor  
24543 Jps
```

One worker process is assigned to the supervisor-2 machine.

A total of three worker processes are running on the Storm cluster.

Let's try reconfiguring `SampleStormClusterTopology` to use two worker processes, `SampleSpout` to use four executors, and `SampleBolt` to use four executors:

```
> bin/storm rebalance SampleStormClusterTopology -n 2 -e SampleSpout=4 -e
  SampleBolt=4
  0      [main] INFO  backtype.storm.thrift  - Connecting to Nimbus at
  nimbus.host.ip:6627
  58     [main] INFO  backtype.storm.command.rebalance  - Topology
  SampleStormClusterTopology is rebalancing
```

Rerun the `jps` commands on the supervisor machines to view the number of worker processes.

Run the `jps` command on supervisor-1:

```
> jps
24377 worker
23940 supervisor
24593 Jps
```

Run the `jps` command on supervisor-2:

```
> jps
24353 worker
23941 supervisor
24543 Jps
```

In this case, two worker processes are shown previously. The first worker process is assigned to supervisor-1 and the other one is assigned to supervisor-2. The distribution of workers may vary depending on the number of topologies running on the system and the number of slots available on each supervisor. Ideally, Storm tries to distribute the load uniformly between all the nodes.

Different types of stream grouping in the Storm cluster

When defining a topology, we create a graph of computation with the number of bolt-processing streams. At a more granular level, each bolt executes multiple tasks in the topology. Thus, each task of a particular bolt will only get a subset of the tuples from the subscribed streams.

Stream grouping in Storm provides complete control over how this partitioning of tuples happens among the many tasks of a bolt subscribed to a stream. Grouping for a bolt can be defined on the instance of `org.apache.storm.topology.InputDeclarer` returned when defining bolts using the `org.apache.e.storm.topology.TopologyBuilder.setBolt` method.

Storm supports the following types of stream groupings.

Shuffle grouping

Shuffle grouping distributes tuples in a uniform, random way across the tasks. An equal number of tuples will be processed by each task. This grouping is ideal when you want to distribute your processing load uniformly across the tasks and where there is no requirement for any data-driven partitioning. This is one of the most commonly used groupings in Storm.

Field grouping

This grouping enables you to partition a stream on the basis of some of the fields in the tuples. For example, if you want all the tweets from a particular user to go to a single task, then you can partition the tweet stream using field grouping by username in the following manner:

```
builder.setSpout("1", new TweetSpout());  
builder.setBolt("2", new TweetCounter()).fieldsGrouping("1", new  
Fields("username"))
```

As a result of the field grouping being $\text{hash}(\text{fields}) \% (\text{no. of tasks})$, it does not guarantee that each of the tasks will get tuples to process. For example, if you have applied a field grouping on a field, say X , with only two possible values, A and B , and created two tasks for the bolt, then it might be possible that both $\text{hash}(A) \% 2$ and $\text{hash}(B) \% 2$ return equal values, which will result in all the tuples being routed to a single task and the other being completely idle.

Another common usage of field grouping is to join streams. Since partitioning happens solely on the basis of field values, and not the stream type, we can join two streams with any common join fields. The name of the fields needs not be the same. For example, in the order processing domain, we can join the `Order` stream and the `ItemScanned` stream to see when an order is completed:

```
builder.setSpout("1", new OrderSpout());
builder.setSpout("2", new ItemScannedSpout());
builder.setBolt("joiner", new OrderJoiner())
    .fieldsGrouping("1", new Fields("orderId"))
    .fieldsGrouping("2", new Fields("orderRefId"));
```

Since joins on streams vary from application to application, you'll make your own definition of a join, say joins over a time window, that can be achieved by composing field groupings.

All grouping

All grouping is a special grouping that does not partition the tuples but replicates them to all the tasks, that is, each tuple will be sent to each of the bolt's tasks for processing.

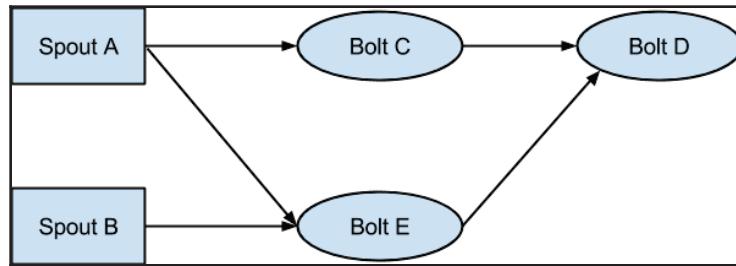
One common use case of all grouping is for sending signals to bolts. For example, if you are doing some kind of filtering on the streams, you can pass or change the filter parameters to all the bolts by sending them those parameters over a stream that is subscribed by all the bolt's tasks with an all grouping. Another example is to send a reset message to all the tasks in an aggregation bolt.

Global grouping

Global grouping does not partition the stream but sends the complete stream to the bolt's task, the smallest ID. A general use case of this is when there needs to be a reduce phase in your topology where you want to combine the results from previous steps in the topology into a single bolt.

Global grouping might seem redundant at first, as you can achieve the same results by defining the parallelism for the bolt as one if you only have one input stream. However, when you have multiple streams of data coming through a different path, you might want only one of the streams to be reduced and others to be parallel processes.

For example, consider the following topology. In this, you might want to combine all the tuples coming from **Bolt C** in a single **Bolt D** task, while you might still want parallelism for tuples coming from **Bolt E** to **Bolt D**:



Direct grouping

In direct grouping, the emitter decides where each tuple will go for processing. For example, say we have a log stream and we want to process each log entry to be processed by a specific bolt task on the basis of the type of resource. In this case, we can use direct grouping.

Direct grouping can only be used with direct streams. To declare a stream as a direct stream, use the `backtype.storm.topology.OutputFieldsDeclarer.declareStream` method, which takes a boolean parameter. Once you have a direct stream to emit to, use `backtype.storm.task.OutputCollector.emitDirect` instead of `emit` methods to emit it. The `emitDirect` method takes a `taskId` parameter to specify the task. You can get the number of tasks for a component using the `backtype.storm.task.TopologyContext.getComponentTasks` method.

Local or shuffle grouping

If the tuple source and target bolt tasks are running in the same worker, using this grouping will act as a shuffle grouping only between the target tasks running on the same worker, thus minimizing any network hops, resulting in increased performance.

If there are no target bolt tasks running on the source worker process, this grouping will act similar to the shuffle grouping mentioned earlier.

None grouping

None grouping is used when you don't care about the way tuples are partitioned among various tasks. As of Storm 0.8, this is equivalent to using shuffle grouping.

Custom grouping

If none of the preceding groupings fit your use case, you can define your own custom grouping by implementing the `backtype.storm.grouping.CustomStreamGrouping` interface.

Here is a sample custom grouping that partitions the stream on the basis of the category in the tuples:

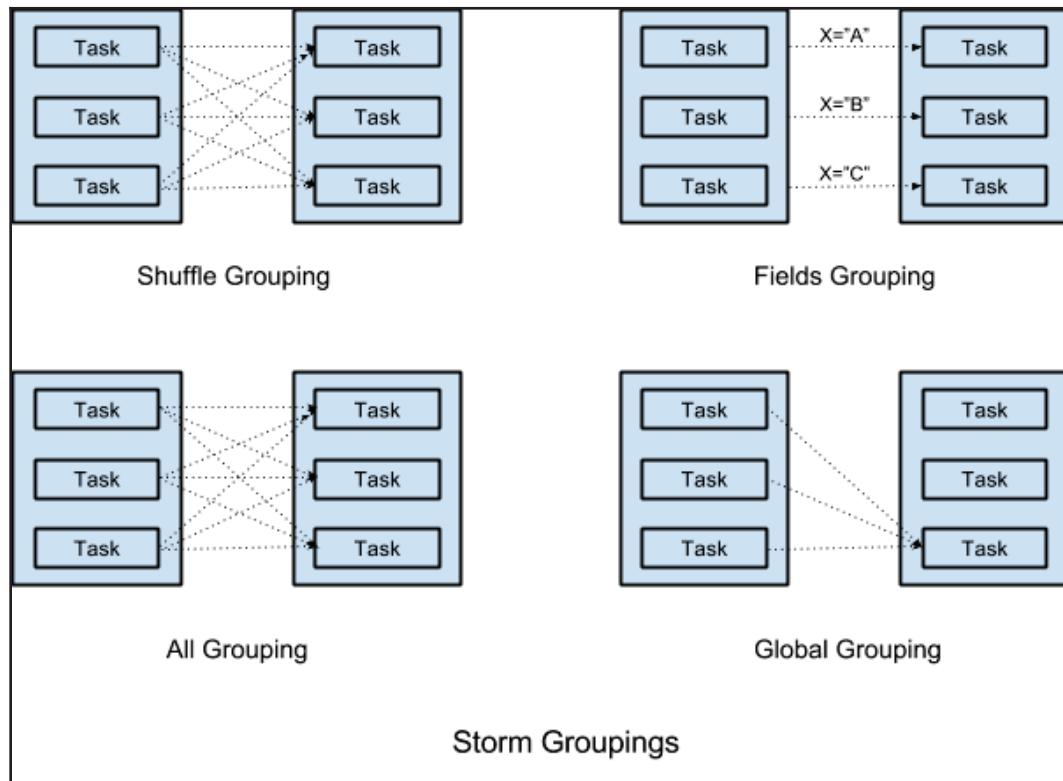
```
public class CategoryGrouping implements CustomStreamGrouping, Serializable
{
    private static final Map<String, Integer> categories = ImmutableMap.of
    (
        "Financial", 0,
        "Medical", 1,
        "FMCG", 2,
        "Electronics", 3
    );

    private int tasks = 0;

    public void prepare(WorkerTopologyContext context, GlobalStreamId stream,
List<Integer> targetTasks)
    {
        tasks = targetTasks.size();
    }

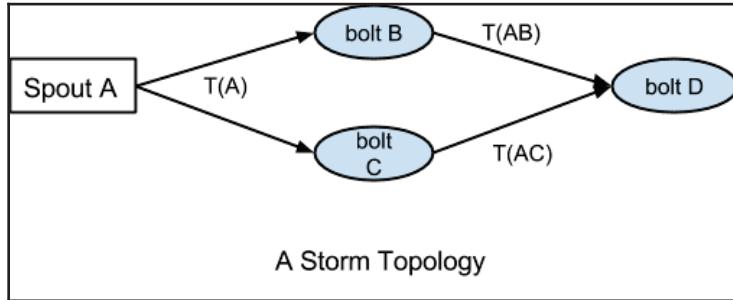
    public List<Integer> chooseTasks(int taskId, List<Object> values) {
        String category = (String) values.get(0);
        return ImmutableList.of(categories.get(category) % tasks);
    }
}
```

The following diagram represents the Storm groupings graphically:



Guaranteed message processing

In a Storm topology, a single tuple being emitted by a spout can result in a number of tuples being generated in the later stages of the topology. For example, consider the following topology:



Here, **Spout A** emits a tuple $T(A)$, which is processed by **bolt B** and **bolt C**, which emit tuple $T(AB)$ and $T(AC)$ respectively. So, when all the tuples produced as a result of tuple $T(A)$ --namely, the tuple tree $T(A)$, $T(AB)$, and $T(AC)$ --are processed, we say that the tuple has been processed completely.

When some of the tuples in a tuple tree fail to process either due to some runtime error or a timeout that is configurable for each topology, then Storm considers that to be a failed tuple.

Here are the six steps that are required by Storm to guarantee message processing:

1. Tag each tuple emitted by a spout with a unique message ID. This can be done by using the `org.apache.storm.spout.SpoutOutputCollector.emit` method, which takes a `messageId` argument. Storm uses this message ID to track the state of the tuple tree generated by this tuple. If you use one of the emit methods that doesn't take a `messageId` argument, Storm will not track it for complete processing. When the message is processed completely, Storm will send an acknowledgement with the same `messageId` that was used while emitting the tuple.
2. A generic pattern implemented by spouts is that they read a message from a messaging queue, say RabbitMQ, produce the tuple into the topology for further processing, and then dequeue the message once it receives the acknowledgement that the tuple has been processed completely.
3. When one of the bolts in the topology needs to produce a new tuple in the course of processing a message, for example, **bolt B** in the preceding topology, then it should emit the new tuple anchored with the original tuple that it got from the spout. This can be done by using the overloaded emit methods in the `org.apache.storm.task.OutputCollector` class that takes an anchor tuple as an argument. If you are emitting multiple tuples from the same input tuple, then anchor each outgoing tuple.

4. Whenever you are done with processing a tuple in the execute method of your bolt, send an acknowledgement using the `org.apache.storm.task.OutputCollector.ack` method. When the acknowledgement reaches the emitting spout, you can safely mark the message as being processed and dequeue it from the message queue, if any.
5. Similarly, if there is some problem in processing a tuple, a failure signal should be sent back using the `org.apache.storm.task.OutputCollector.fail` method so that Storm can replay the failed message.
6. One of the general patterns of processing in Storm bolts is to process a tuple in, emit new tuples, and send an acknowledgement at the end of the execute method. Storm provides the `org.apache.storm.topology.base.BasicBasicBolt` class that automatically sends the acknowledgement at the end of the execute method. If you want to signal a failure, throw `org.apache.storm.topology.FailedException` from the execute method.

This model results in at-least-once message processing semantics, and your application should be ready to handle a scenario when some of the messages will be processed multiple times. Storm also provides exactly-once message processing semantics, which we will discuss in [Chapter 5, Trident Topology and Uses](#).

Even though you can achieve some guaranteed message processing in Storm using the methods mentioned here, it is always a point to ponder whether you actually require it or not, as you can gain a large performance boost by risking some of the messages not being completely processed by Storm. This is a trade-off that you can think of when designing your application.

Tick tuple

In some use cases, a bolt needs to cache the data for a few seconds before performing some operation, such as cleaning the cache after every 5 seconds or inserting a batch of records into a database in a single request.

The tick tuple is the system-generated (Storm-generated) tuple that we can configure at each bolt level. The developer can configure the tick tuple at the code level while writing a bolt.

We need to overwrite the following method in the bolt to enable the tick tuple:

```
@Override  
public Map<String, Object> getComponentConfiguration() {  
    Config conf = new Config();  
    int tickFrequencyInSeconds = 10;  
    conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS,  
        tickFrequencyInSeconds);  
    return conf;  
}
```

In the preceding code, we have configured the tick tuple time to 10 seconds. Now, Storm will start generating a tick tuple after every 10 seconds.

Also, we need to add the following code in the execute method of the bolt to identify the type of tuple:

```
@Override  
public void execute(Tuple tuple) {  
    if (isTickTuple(tuple)) {  
        // now you can trigger e.g. a periodic activity  
    }  
    else {  
        // do something with the normal tuple  
    }  
}  
  
private static boolean isTickTuple(Tuple tuple) {  
    return  
        tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID) &&  
        tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID);  
}
```

If the output of the `isTickTuple()` method is true, then the input tuple is a tick tuple. Otherwise, it is a normal tuple emitted by the previous bolt.



Be aware that tick tuples are sent to bolts/spouts just like regular tuples, which means they will be queued behind other tuples that a bolt/spout is about to process via its `execute()` or `nextTuple()` method, respectively. As such, the time interval you configure for tick tuples is, in practice, served on a best-effort basis. For instance, if a bolt is suffering from high execution latency--for example, due to being overwhelmed by the incoming rate of regular, non-tick tuples--then you will observe that the periodic activities implemented in the bolt will get triggered later than expected.

Summary

In this chapter, we have shed some light on how we can define the parallelism of Storm, how we can distribute jobs between multiple nodes, and how we can distribute data between multiple instances of a bolt. The chapter also covered two important features: guaranteed message processing and the tick tuple.

In the next chapter, we are covering the Trident high-level abstraction over Storm. Trident is mostly used to solve the real-time transaction problem, which can't be solved through plain Storm.

8

Integration of Storm and Kafka

Apache Kafka is a high-throughput, distributed, fault-tolerant, and replicated messaging system that was first developed at LinkedIn. The use cases of Kafka vary from log aggregation, to stream processing, to replacing other messaging systems.

Kafka has emerged as one of the important components of real-time processing pipelines in combination with Storm. Kafka can act as a buffer or feeder for messages that need to be processed by Storm. Kafka can also be used as the output sink for results emitted from Storm topologies.

In this chapter, we will be covering the following topics:

- Kafka architecture--broker, producer, and consumer
- Installation of the Kafka cluster
- Sharing the producer and consumer between Kafka
- Development of Storm topology using Kafka consumer as Storm spout
- Deployment of a Kafka and Storm integration topology

Introduction to Kafka

In this section we are going to cover the architecture of Kafka--broker, consumer, and producer.

Kafka architecture

Kafka has an architecture that differs significantly from other messaging systems. Kafka is a peer to peer system (each node in a cluster has the same role) in which each node is called a **broker**. The brokers coordinate their actions with the help of a ZooKeeper ensemble. The Kafka metadata managed by the ZooKeeper ensemble is mentioned in the section *Sharing ZooKeeper between Storm and Kafka*:

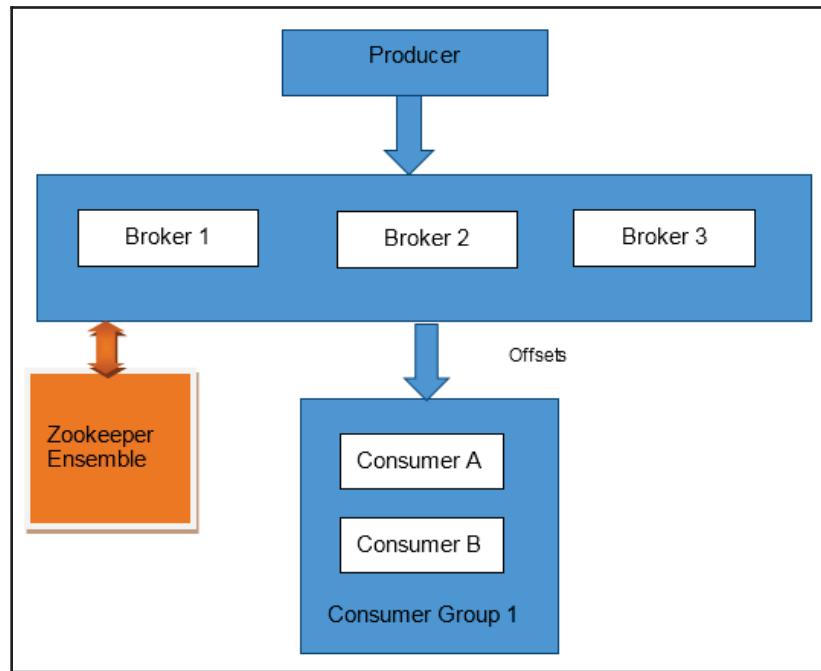


Figure 8.1: A Kafka cluster

The following are the important components of Kafka:

Producer

A producer is an entity that uses the Kafka client API to publish messages into the Kafka cluster. In a Kafka broker, messages are published by the producer entity to named entities called **topics**. A topic is a persistent queue (data stored into topics is persisted to disk).

For parallelism, a Kafka topic can have multiple partitions. Each partition data is represented in a different file. Also, two partitions of a single topic can be allocated on a different broker, thus increasing throughput as all partitions are independent of each other. Each message in a partition has a unique sequence number associated with it called an **offset**:

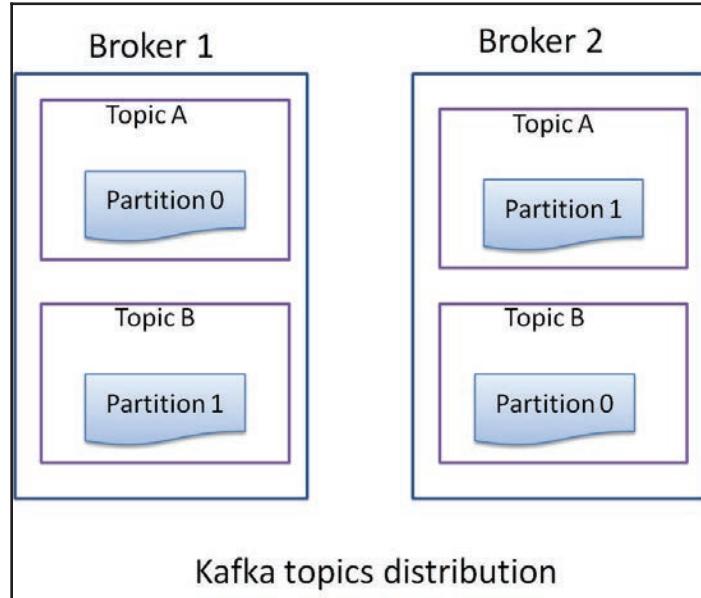


Figure 8.2: Kafka topic distribution

Replication

Kafka supports the replication of partitions of topics to support fault tolerance. Kafka automatically handles the replication of partitions and makes sure that the replica of a partition will be assigned to different brokers. Kafka elects one broker as the leader of a partition and all writes and reads must go to the partition leader. Replication features are introduced in Kafka 8.0.0 version.

The Kafka cluster manages the list of **in sync replica (ISR)**--the replicate which are in sync with the partition leader into ZooKeeper. If the partition leader goes down, then the followers/replicas that are present in the ISR list are only eligible for the next leader of the failed partition.

Consumer

Consumers read a range of messages from a broker. Each consumer has an assigned group ID. All the consumers with the same group ID act as a single logical consumer. Each message of a topic is delivered to one consumer from a consumer group (with the same group ID). Different consumer groups for a particular topic can process messages at their own pace as messages are not removed from the topics as soon as they are consumed. In fact, it is the responsibility of the consumers to keep track of how many messages they have consumed.

As mentioned earlier, each message in a partition has a unique sequence number associated with it called an offset. It is through this offset that consumers know how much of the stream they have already processed. If a consumer decides to replay already processed messages, all it needs to do is just set the value of an offset to an earlier value before consuming messages from Kafka.

Broker

The broker receives the messages from the producer (push mechanism) and delivers the messages to the consumer (pull mechanism). Brokers also manage the persistence of messages in a file. Kafka brokers are very lightweight: they only open file pointers on a queue (topic partitions) and manage TCP connections.

Data retention

Each topic in Kafka has an associated retention time. When this time expires, Kafka deletes the expired data file for that particular topic. This is a very efficient operation as it's a file delete operation.

Installation of Kafka brokers

At the time of writing, the stable version of Kafka is 0.9.x.

The prerequisites for running Kafka are a ZooKeeper ensemble and Java Version 1.7 or above. Kafka comes with a convenience script that can start a single node ZooKeeper but it is not recommended to use it in a production environment. We will be using the ZooKeeper cluster we deployed in [Chapter 2, Storm Deployment, Topology Development, and Topology Options](#).

We will see how to set up a single node Kafka cluster first and then how to add two more nodes to it to run a full-fledged, three node Kafka cluster with replication enabled.

Setting up a single node Kafka cluster

Following are the steps to set up a single node Kafka cluster:

1. Download the Kafka 0.9.x binary distribution named `kafka_2.10-0.9.0.1.tar.gz` from http://apache.claz.org/kafka/0.9.0.1/kafka_2.10-0.9.0.1.tgz.
2. Extract the archive to wherever you want to install Kafka with the following command:

```
tar -xvzf kafka_2.10-0.9.0.1.tgz  
cd kafka_2.10-0.9.0.1
```

We will refer to the Kafka installation directory as `$KAFKA_HOME` from now on.

3. Change the following properties in the `$KAFKA_HOME/config/server.properties` file:

```
log.dirs=/var/kafka-  
logszookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
```

Here, `zoo1`, `zoo2`, and `zoo3` represent the hostnames of the ZooKeeper nodes.

The following are the definitions of the important properties in the `server.properties` file:

- `broker.id`: This is a unique integer ID for each of the brokers in a Kafka cluster.
- `port`: This is the port number for a Kafka broker. Its default value is 9092. If you want to run multiple brokers on a single machine, give a unique port to each broker.
- `host.name`: The hostname to which the broker should bind and advertise itself.

- `log.dirs`: The name of this property is a bit unfortunate as it represents not the log directory for Kafka, but the directory where Kafka stores the actual data sent to it. This can take a single directory or a comma-separated list of directories to store data. Kafka throughput can be increased by attaching multiple physical disks to the broker node and specifying multiple data directories, each lying on a different disk. It is not much use specifying multiple directories on the same physical disk, as all the I/O will still be happening on the same disk.
- `num.partitions`: This represents the default number of partitions for newly created topics. This property can be overridden when creating new topics. A greater number of partitions results in greater parallelism at the cost of a larger number of files.
- `log.retention.hours`: Kafka does not delete messages immediately after consumers consume them. It retains them for the number of hours defined by this property so that in the event of any issues the consumers can replay the messages from Kafka. The default value is 168 hours, which is 1 week.
- `zookeeper.connect`: This is the comma-separated list of ZooKeeper nodes in `hostname:port` form.

4. Start the Kafka server by running the following command:

```
> ./bin/kafka-server-start.sh config/server.properties
[2017-04-23 17:44:36,667] INFO New leader is 0
(kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
[2017-04-23 17:44:36,668] INFO Kafka version : 0.9.0.1
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-23 17:44:36,668] INFO Kafka commitId : a7a17cdec9eaa6c5
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-23 17:44:36,670] INFO [Kafka Server 0], started
(kafka.server.KafkaServer)
```

If you get something similar to the preceding three lines on your console, then your Kafka broker is up-and-running and we can proceed to test it.

5. Now we will verify that the Kafka broker is set up correctly by sending and receiving some test messages. First, let's create a verification topic for testing by executing the following command:

```
> bin/kafka-topics.sh --zookeeper zoo1:2181 --replication-factor 1  
--partition 1 --topic verification-topic --create  
Created topic "verification-topic".
```

6. Now let's verify if the topic creation was successful by listing all the topics:

```
> bin/kafka-topics.sh --zookeeper zoo1:2181 --list  
verification-topic
```

7. The topic is created; let's produce some sample messages for the Kafka cluster. Kafka comes with a command-line producer that we can use to produce messages:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --  
topic verification-topic
```

8. Write the following messages on your console:

```
Message 1  
Test Message 2  
Message 3
```

9. Let's consume these messages by starting a new console consumer on a new console window:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic  
verification-topic --from-beginning  
Message 1  
Test Message 2  
Message 3
```

Now, if we enter any message on the producer console, it will automatically be consumed by this consumer and displayed on the command line.



Using Kafka's single node ZooKeeper

If you don't want to use an external ZooKeeper ensemble, you can use the single node ZooKeeper instance that comes with Kafka for quick and dirty development. To start using it, first modify the

`$KAFKA_HOME/config/zookeeper.properties` file to specify the data directory by supplying following property:

`dataDir=/var/zookeeper`

Now, you can start the Zookeeper instance with the following command:

```
> ./bin/zookeeper-server-start.sh  
config/zookeeper.properties
```

Setting up a three node Kafka cluster

So far we have a single node Kafka cluster. Follow the steps to deploy the Kafka cluster:

1. Create a three node VM or three physical machines.
2. Perform steps 1 and 2 mentioned in the section *Setting up a single node Kafka cluster*.
3. Change the following properties in the file

`$KAFKA_HOME/config/server.properties`:

```
broker.id=0  
port=9092  
host.name=kafka1  
log.dirs=/var/kafka-logs  
zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
```

Make sure that the value of the `broker.id` property is unique for each Kafka broker and the value of `zookeeper.connect` must be the same on all nodes.

4. Start the Kafka brokers by executing the following command on all three boxes:

```
> ./bin/kafka-server-start.sh config/server.properties
```

5. Now let's verify the setup. First we create a topic with the following command:

```
> bin/kafka-topics.sh --zookeeper zoo1:2181 --replication-factor 4  
--partition 1 --topic verification --create  
Created topic "verification-topic".
```

6. Now, we will list the topics to see if the topic was created successfully:

```
> bin/kafka-topics.sh --zookeeper zoo1:2181 --list
topic: verification      partition: 0      leader: 0
replicas: 0              isr: 0
topic: verification      partition: 1      leader: 1
replicas: 1              isr: 1
topic: verification      partition: 2      leader: 2
replicas: 2              isr: 2
```

7. Now, we will verify the setup by using the Kafka console producer and consumer as done in the *Setting up a single node Kafka cluster* section:

```
> bin/kafka-console-producer.sh --broker-list
kafka1:9092,kafka2:9092,kafka3:9092 --topic verification
```

8. Write the following messages on your console:

```
First
Second
Third
```

9. Let's consume these messages by starting a new console consumer on a new console window:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic
verification --from-beginning
First
Second
Third
```

So far, we have three brokers on the Kafka cluster working. In the next section, we will see how to write a producer that can produce messages to Kafka:

Multiple Kafka brokers on a single node

If you want to run multiple Kafka brokers on a single node, then follow the following steps:

1. Copy config/server.properties to create config/server1.properties and config/server2.properties.

2. Populate the following properties in config/server.properties:

```
broker.id=0
port=9092
log.dirs=/var/kafka-logs
zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
```

3. Populate the following properties in config/server1.properties:

```
broker.id=1
port=9093
log.dirs=/var/kafka-1-logs
zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
```

4. Populate the following properties in config/server2.properties:

```
broker.id=2
port=9094
log.dirs=/var/kafka-2-logs
zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
```

5. Run the following commands on three different terminals to start Kafka brokers:

```
> ./bin/kafka-server-start.sh config/server.properties
> ./bin/kafka-server-start.sh config/server1.properties
> ./bin/kafka-server-start.sh config/server2.properties
```

Share ZooKeeper between Storm and Kafka

We can share the same ZooKeeper ensemble between Kafka and Storm as both store the metadata inside the different znodes (ZooKeeper coordinates between the distributed processes using the shared hierarchical namespace, which is organized similarly to a standard file system. In ZooKeeper, the namespace consisting of data registers is called znodes).

We need to open the ZooKeeper client console to view the znodes (shared namespace) created for Kafka and Storm.

Go to ZK_HOME and execute the following command to open the ZooKeeper console:

```
> bin/zkCli.sh
```

Execute the following command to view the list of znodes:

```
> [zk: localhost:2181(CONNECTED) 0] ls /
[storm, consumers, isr_change_notification, zookeeper, admin, brokers]
```

Here, consumers, `isr_change_notification`, and brokers are the znodes and the Kafka is managing its metadata information into ZooKeeper at this location.

Storm manages its metadata inside the Storm znodes in ZooKeeper.

Kafka producers and publishing data into Kafka

In this section we are writing a Kafka producer that will publish events into the Kafka topic.

Perform the following step to create the producer:

1. Create a Maven project by using `com.stormadvance` as groupId and `kafka-producer` as artifactId.
2. Add the following dependencies for Kafka in the `pom.xml` file:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.10</artifactId>
    <version>0.9.0.1</version>
    <exclusions>
        <exclusion>
            <groupId>com.sun.jdmk</groupId>
            <artifactId>jmxtools</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jmx</groupId>
            <artifactId>jmxri</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.0-beta9</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
```

```
<version>2.0-beta9</version>
</dependency>
```

3. Add the following build plugins to the pom.xml file. It will let us execute the producer using Maven:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.2.1</version>
      <executions>
        <execution>
          <goals>
            <goal>exec</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <executable>java</executable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4. Now we will create the KafkaSampleProducer class in the com.stormadvance.kafka_producer package. This class will produce each word from the first paragraph of Franz Kafka's Metamorphosis into the new_topic topic in Kafka as single message. The following is the code for the KafkaSampleProducer class with explanations:

```
public class KafkaSampleProducer {
  public static void main(String[] args) {
    // Build the configuration required for connecting to Kafka
    Properties props = new Properties();

    // List of kafka borkers. Complete list of brokers is not
    // required as
    // the producer will auto discover the rest of the brokers.
    props.put("bootstrap.servers", "Broker1-IP:9092");
```

```
        props.put("batch.size", 1);
        // Serializer used for sending data to kafka. Since we are
        sending string,
        // we are using StringSerializer.
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        props.put("producer.type", "sync");
        // Create the producer instance
        Producer<String, String> producer = new KafkaProducer<String,
String>(props);

        // Now we break each word from the paragraph
        for (String word : METAMORPHOSIS_OPENING_PARA.split("\\s")) {
            System.out.println("word : " + word);
            // Create message to be sent to "new_topic" topic with the
            word
            ProducerRecord<String, String> data = new
            ProducerRecord<String, String>("new_topic", word, word);
            // Send the message
            producer.send(data);
        }

        // close the producer
        producer.close();
        System.out.println("end : ");
    }

    // First paragraph from Franz Kafka's Metamorphosis
    private static String METAMORPHOSIS_OPENING_PARA = "One morning,
when Gregor Samsa woke from troubled dreams, he found "
        + "himself transformed in his bed into a horrible
vermin. He lay on "
        + "his armour-like back, and if he lifted his head a
little he could "
        + "see his brown belly, slightly domed and divided
by arches into stiff "
        + "sections. The bedding was hardly able to cover
it and seemed ready "
        + "to slide off any moment. His many legs,
pitifully thin compared "
        + "with the size of the rest of him, waved about
helplessly as he "
        + "looked.";

}
```

5. Now, before running the producer, we need to create new_topic in Kafka. To do so, execute the following command:

```
> bin/kafka-topics.sh --zookeeper ZK1:2181 --replication-factor 1 --partition 1 --topic new_topic --create  
Created topic "new_topic1".
```

6. Now we can run the producer by executing the following command:

```
> mvn compile exec:java  
.....  
103 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.client.ClientUti  
ls$ - Fetching metadata from broker  
id:0,host:kafka1,port:9092 with correlation id 0 for 1  
topic(s) Set(words_topic)  
110 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.producer.SyncProducer - Connected to kafka1:9092 for  
producing  
140 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.producer.SyncProducer - Disconnecting from  
kafka1:9092  
177 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.producer.SyncProducer - Connected to kafka1:9092 for  
producing  
378 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.producer.Producer - Shutting down producer  
378 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.producer.ProducerPool - Closing all sync producers  
381 [com.learningstorm.kafka.WordsProducer.main()] INFO  
kafka.producer.SyncProducer - Disconnecting from  
kafka1:9092
```

7. Now let us verify that the message has been produced by using Kafka's console consumer and executing the following command:

```
> bin/kafka-console-consumer.sh --zookeeper ZK:2181 --topic verification --from-beginning  
One  
morning,  
when  
Gregor  
Samsa  
woke  
from  
troubled  
dreams,
```

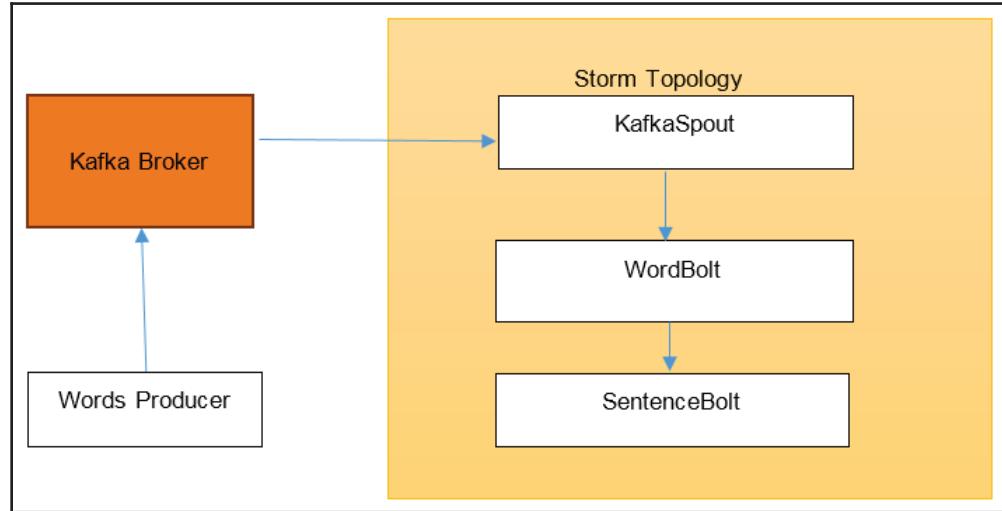
```
he
found
himself
transformed
in
his
bed
into
a
horrible
vermin.
.....
```

So, we are able to produce messages into Kafka. In the next section, we will see how we can use KafkaSpout to read messages from Kafka and process them inside a Storm topology.

Kafka Storm integration

Now we will create a Storm topology that will consume messages from the Kafka topic new_topic and aggregate words into sentences.

The complete message flow is shown as follows:



We have already seen `KafkaSampleProducer`, which produces words into the Kafka broker. Now we will create a Storm topology that will read those words from Kafka to aggregate them into sentences. For this, we will have one `KafkaSpout` in the application that will read the messages from Kafka and two bolts, `WordBolt` that receive words from `KafkaSpout` and then aggregate them into sentences, which are then passed onto the `SentenceBolt`, which simply prints them on the output stream. We will be running this topology in a local mode.

Follow the steps to create the Storm topology:

1. Create a new Maven project with groupId as `com.stormadvance` and artifactId as `kafka-storm-topology`.
2. Add the following dependencies for Kafka-Storm and Storm in the `pom.xml` file:

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-kafka</artifactId>
    <version>1.0.2</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.10</artifactId>
    <version>0.9.0.1</version>
    <exclusions>
        <exclusion>
            <groupId>com.sun.jdmk</groupId>
            <artifactId>jmxtools</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jmx</groupId>
            <artifactId>jmxri</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.0.2</version>
```

```
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>commons-collections</groupId>
        <artifactId>commons-collections</artifactId>
        <version>3.2.1</version>
    </dependency>

    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
        <version>15.0</version>
    </dependency>
```

3. Add the following Maven plugins to the `pom.xml` file so that we are able to run it from the command-line and also to package the topology to be executed in Storm:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
                <archive>
                    <manifest>
                        <mainClass></mainClass>
                    </manifest>
                </archive>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.2.1</version>
            <executions>
                <execution>
```

```
<goals>
    <goal>exec</goal>
</goals>
</execution>
</executions>
<configuration>
    <executable>java</executable>
<includeProjectDependencies>true</includeProjectDependencies>
<includePluginDependencies>false</includePluginDependencies>
    <classpathScope>compile</classpathScope>
    <mainClass>${main.class}</mainClass>
</configuration>
</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
</plugin>

</plugins>
</build>
```

4. Now we will first create the WordBolt that will aggregate the words into sentences. For this, create a class called WordBolt in the com.stormadvance.kafka package. The code for WordBolt is as follows, complete with explanation:

```
public class WordBolt extends BaseBasicBolt {

    private static final long serialVersionUID =
-5353547217135922477L;

    // list used for aggregating the words
    private List<String> words = new ArrayList<String>();

    public void execute(Tuple input, BasicOutputCollector collector)
    {
        System.out.println("called");
        // Get the word from the tuple
        String word = input.getString(0);

        if (StringUtils.isBlank(word)) {
            // ignore blank lines
            return;
        }

        System.out.println("Received Word:" + word);
```

```
// add word to current list of words
words.add(word);

if (word.endsWith(".")) {
    // word ends with '.' which means this is // the end of the
    sentence
    // publish a sentence tuple
    collector.emit(ImmutableList.of((Object)
StringUtils.join(words, ' ')));

    // reset the words list.
    words.clear();
}
}

public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // here we declare we will be emitting tuples with
    // a single field called "sentence"
    declarer.declare(new Fields("sentence"));
}
}
```

5. Next is SentenceBolt, which just prints the sentences that it receives. Create SentenceBolt in the com.stormadvance.kafka package. The code is as follows, with explanations:

```
public class SentenceBolt extends BaseBasicBolt {

    private static final long serialVersionUID =
7104400131657100876L;

    public void execute(Tuple input, BasicOutputCollector collector)
{
    // get the sentence from the tuple and print it
    System.out.println("Recieved Sentence:");
    String sentence = input.getString(0);
    System.out.println("Recieved Sentence:" + sentence);
}

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // we don't emit anything
    }
}
```

6. Now we will create the KafkaTopology that will define the KafkaSpout and wire it with WordBolt and SentenceBolt. Create a new class called KafkaTopology in the com.stormadvance.kafka package. The code is as follows, with explanations:

```
public class KafkaTopology {  
    public static void main(String[] args) {  
        try {  
            // ZooKeeper hosts for the Kafka cluster  
            BrokerHosts zkHosts = new ZkHosts("ZKIP:PORT");  
  
            // Create the KafkaSpout configuartion  
            // Second argument is the topic name  
            // Third argument is the zookeepr root for Kafka  
            // Fourth argument is consumer group id  
            SpoutConfig kafkaConfig = new SpoutConfig(zkHosts,  
                "new_topic", "", "id1");  
  
            // Specify that the kafka messages are String  
            // We want to consume all the first messages in the topic  
            // everytime  
            // we run the topology to help in debugging. In production,  
            // this  
            // property should be false  
            kafkaConfig.scheme = new SchemeAsMultiScheme(new  
                StringScheme());  
            kafkaConfig.startOffsetTime =  
                kafka.api.OffsetRequest.EarliestTime();  
  
            // Now we create the topology  
            TopologyBuilder builder = new TopologyBuilder();  
  
            // set the kafka spout class  
            builder.setSpout("KafkaSpout", new KafkaSpout(kafkaConfig),  
                2);  
  
            // set the word and sentence bolt class  
            builder.setBolt("WordBolt", new WordBolt(),  
                1).globalGrouping("KafkaSpout");  
            builder.setBolt("SentenceBolt", new SentenceBolt(),  
                1).globalGrouping("WordBolt");  
  
            // create an instance of LocalCluster class for executing  
            // topology  
            // in local mode.  
            LocalCluster cluster = new LocalCluster();  
            Config conf = new Config();
```

```
conf.setDebug(true);
if (args.length > 0) {
    conf.setNumWorkers(2);
    conf.setMaxSpoutPending(5000);
    StormSubmitter.submitTopology("KafkaTopology1", conf,
builder.createTopology());

} else {
    // Submit topology for execution
    cluster.submitTopology("KafkaTopology1", conf,
builder.createTopology());
    System.out.println("called1");
    Thread.sleep(1000000);
    // Wait for sometime before exiting
    System.out.println("Waiting to consume from kafka");

    System.out.println("called2");
    // kill the KafkaTopology
    cluster.killTopology("KafkaTopology1");
    System.out.println("called3");
    // shutdown the storm test cluster
    cluster.shutdown();
}

} catch (Exception exception) {
    System.out.println("Thread interrupted exception : " +
exception);
}
}
```

7. Now we will run the topology. Make sure the Kafka cluster is running and you have executed the producer in the last section so that there are messages in Kafka for consumption.
8. Run the topology by executing the following command:

```
> mvn clean compile exec:java -
Dmain.class=com.stormadvance.kafka.KafkaTopology
```

This will execute the topology. You should see messages similar to the following in your output:

```
Recieved Word:One
Recieved Word:morning,
Recieved Word:when
Recieved Word:Gregor
Recieved Word:Samsa
```

```
Recieved Word:woke
Recieved Word:from
Recieved Word:troubled
Recieved Word:dreams,
Recieved Word:he
Recieved Word:found
Recieved Word:himself
Recieved Word:transformed
Recieved Word:in
Recieved Word:his
Recieved Word:bed
Recieved Word:into
Recieved Word:a
Recieved Word:horrible
Recieved Word:vermin.
Recieved Sentence:One morning, when Gregor Samsa woke from
troubled dreams, he found himself transformed in his bed
into a horrible vermin.
```

So we are able to consume messages from Kafka and process them in a Storm topology.

Deploy the Kafka topology on Storm cluster

The deployment of Kafka and Storm integration topology on the Storm cluster is similar to the deployment of other topologies. We need to set the number of workers and the maximum spout pending Storm config and we need to use the submitTopology method of StormSubmitter to submit the topology on the Storm cluster.

Now, we need to build the topology code as mentioned in the following steps to create a JAR of the Kafka Storm integration topology:

1. Go to project home.
2. Execute the command:

```
mvn clean install
```

The output of the preceding command is as follows:

```
-----
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----
```

```
[INFO] Total time: 58.326s
[INFO] Finished at:
[INFO] Final Memory: 14M/116M
[INFO] -----
-----
```

3. Now, copy the Kafka Storm topology on the Nimbus machine and execute the following command to submit the topology on the Storm cluster:

```
bin/storm jar jarName.jar [TopologyMainClass] [Args]
```

The preceding command runs `TopologyMainClass` with the argument. The main function of `TopologyMainClass` is to define the topology and submit it to Nimbus. The Storm JAR part takes care of connecting to Nimbus and uploading the JAR part.

4. Log in on the Storm Nimbus machine and execute the following commands:

```
$> cd $STORM_HOME
$> bin/storm jar ~/storm-kafka-topology-0.0.1-SNAPSHOT-jar-with-
dependencies.jar com.stormadvance.kafka.KafkaTopology
KafkaTopology1
```

Here, `~/storm-kafka-topology-0.0.1-SNAPSHOT-jar-with-dependencies.jar` is the path of the `KafkaTopology` JAR we are deploying on the Storm cluster.

Summary

In this chapter, we learned about the basics of Apache Kafka and how to use it as part of a real-time stream processing pipeline build with Storm. We learned about the architecture of Apache Kafka and how it can be integrated into Storm processing by using `KafkaSpout`.

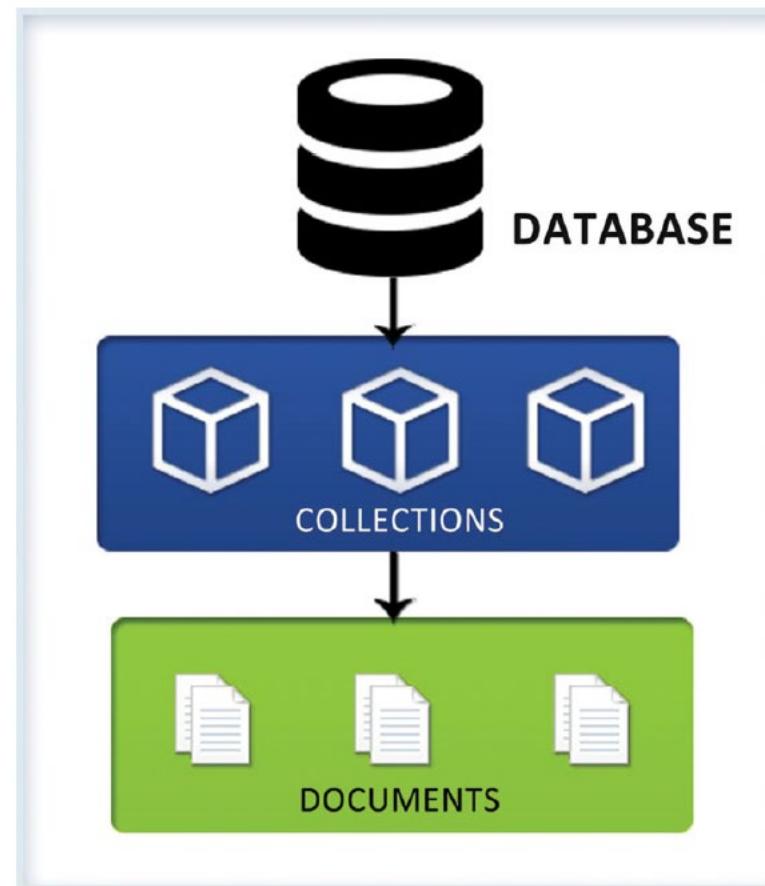
In the next chapter, we are going to cover the integration of Storm with Hadoop and YARN. We are also going to cover sample examples for this operation.

Unit 5 - MongoDB

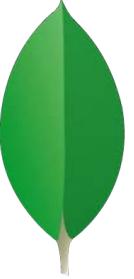
MongoDB Introduction

- 2007, Dwight Merriman, Eliot Horowitz decided to build a database that would not comply with the RDBMS model
- Design Philosophy
 - Speed, Scalability, and Agility
 - Non-Relational Approach
 - JSON-Based Document Store
 - Performance vs. Features
 - Running the Database Anywhere

MongoDB Data Model



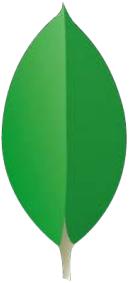
Database



Database

- Made up of Multiple **Collections**.
- Created **on-the-fly** when referenced for the first time.

Collection



Table

- Schema-less, and contains **Documents**.
- **Indexable** by one/more keys.
- Created **on-the-fly** when referenced for the first time.
- **Capped Collections:** Fixed size, older records get dropped after reaching the limit.

Document



Record/Row

- Stored in a **Collection**.
- Can have **_id** key – works like Primary keys in MySQL.
- Supported Relationships – **Embedded (or) References**.
- Document storage in **BSON** (Binary form of JSON).

Understanding the Document Model.

```
var p = {  
  '_id': '3432',  
  'author': DBRef('User', 2),  
  'title': 'Introduction to MongoDB',  
  'body': 'MongoDB is an open sources.. ',  
  'timestamp': Date('01-04-12'),  
  'tags': ['MongoDB', 'NoSQL'],  
  'comments': [{  
    'author': DBRef('User', 4),  
    'date': Date('02-04-12'),  
    'text': 'Did you see.. ',  
    'upvotes': 7, ... }]  
}  
> db.posts.save(p) //p is a document;
```



MongoDB-Data Model-Embedded

- Embedded/De-normalized
- can have (embed) all the related data in a single document

```
{  
    _id: ,  
    Emp_ID: "10025AE336"  
    Personal_details:{  
        First_Name: "Radhika",  
        Last_Name: "Sharma",  
        Date_Of_Birth: "1995-09-26"  
    },  
    Contact: {  
        e-mail: "radhika_sharma.123@gmail.com",  
        phone: "9848022338"  
    },  
    Address: {  
        city: "Hyderabad",  
        Area: "Madapur",  
        State: "Telangana"  
    }  
}
```

MongoDB-Data Model-Normalized

- refer the sub documents in the original document, using references

Employee:

```
{  
  _id: <ObjectId101>,  
  Emp_ID: "10025AE336"  
}
```

Personal_details:

```
{  
  _id: <ObjectId102>,  
  empDocID: " ObjectId101",  
  First_Name: "Radhika",  
  Last_Name: "Sharma",  
  Date_Of_Birth: "1995-09-26"  
}
```

Contact:

```
{  
  _id: <ObjectId103>,  
  empDocID: " ObjectId101",  
  e-mail: "radhika_sharma.123@gmail.com",  
  phone: "9848022338"  
}
```

Address:

```
{  
  _id: <ObjectId104>,  
  empDocID: " ObjectId101",  
  city: "Hyderabad",  
  Area: "Madapur",  
  State: "Telangana"  
}
```

Installation

- MongoDB Community Server Edition for Windows10 64 bit
- FIRST RUN COMMANDS
- -----
- #Step 1 - Create a data folder
- C:\>md data
- C:\>md data\db
- #Step 2 - Specify this data path to the mongodb.exe
- C:\Users\XYZ> cd C:\Program Files\MongoDB\Server\4.2\bin
- C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"
- #This should show waiting for connections
- #Step 3 - Run mongodb.exe from another command prompt
- C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe
- FROM NEXT RUN ONWARDS JUST EXECUTE
- -----
- #Two different command prompt
- C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"
- C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe

Commands – Create database

- Create database:

- Syntax:

```
use DATABASE_NAME
```

- Example:

```
use sample
```

- Output:

```
> use sample
switched to db sample
```

Commands – Show databases

Currently selected database

>db

List of databases – Empty dbs are not listed

>show dbs

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

```
> db
sample
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
```

Commands – Create collections

Syntax:

```
db.createCollection(name, options)
```

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Example:

```
> db.createCollection("books")
{ "ok" : 1 }
>
```

Commands - Collections

- Check Created collections:

```
show collections
```

```
> show collections  
books
```

```
> db.createCollection("players", { capped : true, size : 6142800, max : 10000 } )  
{ "ok" : 1 }
```

```
> db.movies.insert({ "name" : "chamber of secrets" })  
WriteResult({ "nInserted" : 1 })
```

```
> show collections  
books  
movies  
players
```

Some of the data types supported

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.

Commands – Insert document

- Syntax
- db.COLLECTION_NAME.insert(document)
- Single and multiple inserts possible – insertone vs insertmany
- Save is the other alternate for insert

Commands – Query document

- Syntax
- >db.COLLECTION_NAME.find()
- Formatted Display
- >db.COLLECTION_NAME.find().pretty()
- Return any one document
- >db.COLLECTIONNAME.findOne()
- AND OR where clause is possible

Commands – Update document

- Syntax

```
>db.COLLECTION_NAME.update(SELECTION_CRITERIA,  
    UPDATED_DATA)
```

- By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.
- >db.books.update({'title':'MongoDB Overview'}, {\$set:{'title':'New MongoDB Tutorial'}},{multi:true})

Commands – Delete document

- Syntax

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

- If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

```
> db.books.remove({}) WriteResult({ "nRemoved" : 2 }) >  
db.mycol.find() >
```

Mongo DB Data Model Contd

- MongoDB is a document-based database. It uses Binary JSON for storing its data

```
{  
  "_id" : 1,  
  "name" : { "first" : "John", "last" : "Doe" },  
  "publications" : [  
    {  
      "title" : "First Book",  
      "year" : 1989,  
      "publisher" : "publisher1"  
    },  
    { "title" : "Second Book",  
      "year" : 1999,  
      "publisher" : "publisher2"  
    }  
  ]  
}
```

Mongo DB Data Model Contd

- Binary JSON - MongoDB stores the JSON document in a binary-encoded format. This is termed as BSON. The BSON data model is an **extended form of the JSON** data model
- BSON document is **fast**, highly traversable, and lightweight
- Supports **embedding of arrays** and objects within other arrays,
- also enables MongoDB to **reach inside the objects** to build indexes and match objects against queried expressions, both on top-level and nested BSON keys

Mongo DB Data Model Contd `_id` field

- Documents are made up of **key-value pairs**
- Although a document can be compared to a row in RDBMS, unlike a row, documents have **flexible schema**
- A key, which is nothing but a label, can be roughly compared to the column name in RDBMS
- A key is used for querying data from the documents. Hence, like a RDBMS primary key (used to uniquely identify each row), you need to have a key that uniquely identifies each document within a collection. This is referred to as `_id` in MongoDB.

Mongo DB Data Model Contd – Capped Collection

- MongoDB has a concept of capping the collection. This means it stores the documents in the collection in the inserted order.
- As the collection reaches its limit, the documents will be removed from the collection in FIFO (first in, first out) order.
- This means that the least recently inserted documents will be removed first. This is good for use cases where the order of insertion is required to be maintained automatically, and deletion of records after a fixed size is required. One such use cases is log files that get automatically truncated after a certain size.

Mongo DB Data Model Contd – Polymorphic Schemas

- A polymorphic schema is a schema where a collection has documents of different types or schemas.
- A good example of this schema is a collection named Users. Some user documents might have an extra fax number or email address, while others might have only phone numbers, yet all these documents coexist within the same Users collection.
- This schema is generally referred to as a polymorphic schema

MongoDB Architecture

- Core Processes
 - **mongod** , which is the core database process
 - **mongos** , which is the controller and query router for sharded clusters
 - **mongo** , which is the interactive MongoDB shell
- mongod
 - This daemon handles all the data requests, manages the data format, and performs operations for background management.
 - When a mongod is run without any arguments, it connects to the default data directory, which is C:\data\db or /data/db , and default port 27017, where it listens for socket connections.
 - It's important to ensure that the data directory exists and you have write permissions to the directory before the mongod process is started.

MongoDB Architecture

- mongo
 - mongo provides an interactive JavaScript interface for the developer to test queries and operations directly on the database and for the system administrators to manage the database.
 - This is all done via the command line. When the mongo shell is started, it will connect to the default database called test.
 - This database connection value is assigned to global variable db .
 - you need to change the database from test to your database post the first connection is made. You can do this by using <dbname>.

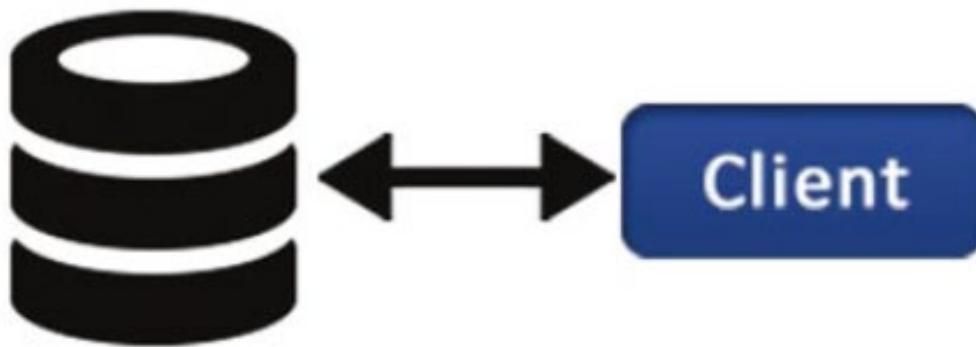
MongoDB Architecture

- **Mongos**
 - mongos is used in MongoDB sharding.
 - It acts as a routing service that processes queries from the application layer and determines where in the sharded cluster the requested data is located.

MongoDB Tools

- **mongodump** : This utility is used as part of an effective **backup** strategy. It creates a binary export of the database contents.
- **mongorestore** : The binary database dump created by the mongodump utility is imported to a ***new or an existing database*** using the mongorestore utility.
- **bsondump** : This utility converts the BSON files into human-readable formats such as JSON and CSV. For example, this utility can be used to read the output file generated by mongodump.
- **mongoimport , mongoexport** : **mongoimport** provides a method for taking data in JSON , CSV, or TSV formats and importing it into a mongod instance. **Mongoexport** provides a method to export data from a mongod instance into JSON, CSV, or TSV formats.
- **mongostat , mongotop , mongosniff** : These utilities provide diagnostic information related to the current operation of a mongod instance.

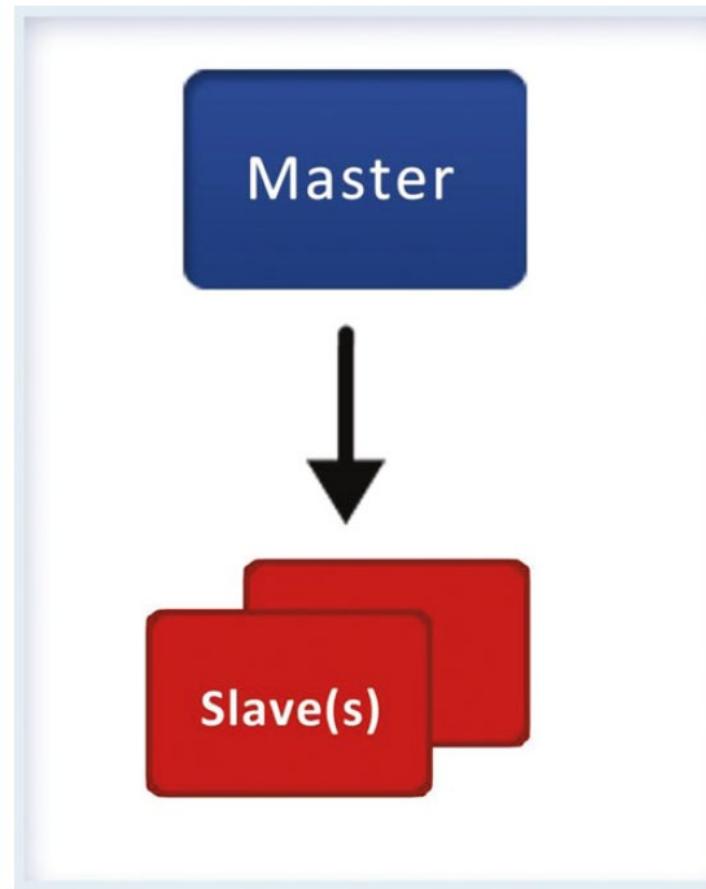
Standalone Deployment



Replication

- Master Slave
- Replication Set

Replication – Master/Slave



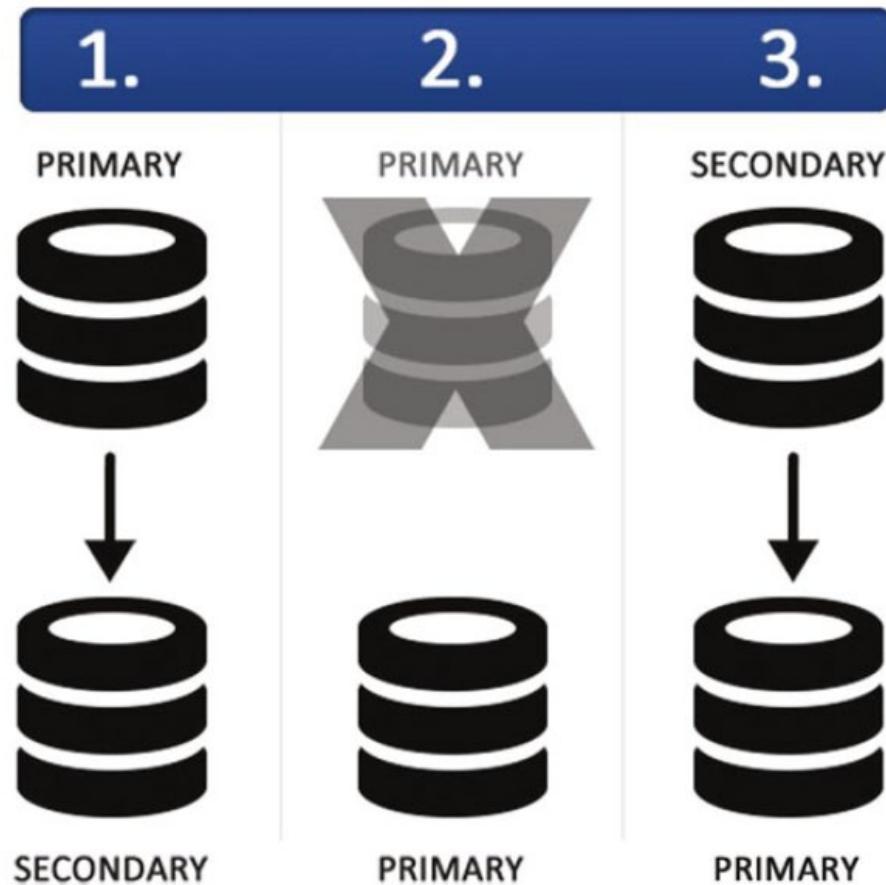
Replication – Master/Slave

- The master node maintains a capped collection (oplog) that stores an ordered history of logical writes to the database.
- The slaves replicate the data using this oplog collection. Since the oplog is a capped collection, if the slave's state is far behind the master's state, the slave may become out of sync.
- In that scenario, the replication will stop and manual intervention will be needed to re-establish the replication.
- There are two main reasons behind a slave becoming out of sync:
 - The slave shuts down or stops and restarts later. During this time, the oplog may have deleted the log of operations required to be applied on the slave.
 - The slave is slow in executing the updates that are available from the master.

Replication – Replica Set

- Replica sets are basically a type of master-slave replication but they provide automatic failover.
- A replica set has one master, which is termed as primary, and multiple slaves, which are termed as secondary in the replica set context;
- However, unlike master-slave replication, there's no one node that is fixed to be primary in the replica set.
- If a master goes down in replica set, automatically one of the slave nodes is promoted to the master.
- The clients start connecting to the new master, and both data and application will remain available. In a replica set, this failover happens in an automated fashion.

Replication – Replica Set



Replication – Replica Set

1. The primary goes down, and the secondary is promoted as primary.
 2. The original primary comes up, it acts as slave, and becomes the secondary node.
- The points to be noted are
 - A replica set is a mongod's cluster, which replicates among one another and ensures automatic failover.
 - In the replica set, one mongod will be the primary member and the others will be secondary members.
 - The primary member is elected by the members of the replica set. All writes are directed to the primary member whereas the secondary members replicate from the primary asynchronously using oplog.
 - The secondary's data sets reflect the primary data sets, enabling them to be promoted to primary in case of unavailability of the current primary.

Replication – Replica Set

- Primary member : A replica set can have only one primary, which is elected by the voting nodes in the replica set. Any node with associated priority as 1 can be elected as a primary. The client redirects all the write operations to the primary member, which is then later replicated to the secondary members.
- Secondary member: A normal secondary member holds the copy of the data. The secondary member can vote and also can be a candidate for being promoted to primary in case of failover of the current primary.

Replication – Replica Set

Types of Secondary:

- Priority 0
- Hidden
- Delayed
- Arbiters
- Non Voting

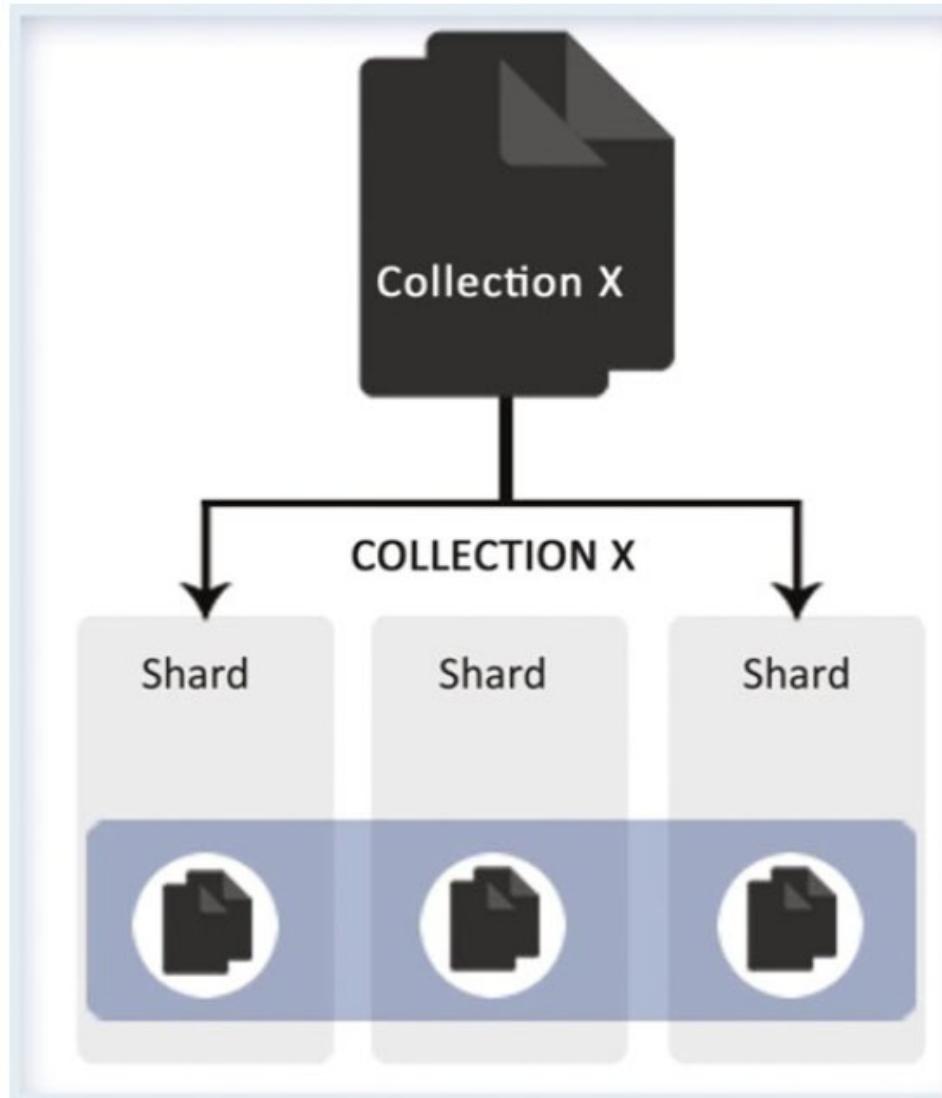
Replication – Replica Set Elections

- If there are X servers with each server having 1 vote, then a server can become primary only when it has at least $[(X/2) + 1]$ votes.
- If a server gets the required number of votes or more, then it will become primary.
- The primary that went down still remains part of the set; when it is up, it will act as a secondary server until the time it gets a majority of votes again.

Sharding in MongoDB

- ideally the working set should fit in memory. The working set consists of the most frequently accessed data and indexes
- the scaling is handled by scaling out the data horizontally, is also called sharding
- Sharding addresses the challenges of scaling to support large data sets and high throughput by horizontally dividing the datasets across servers where each server is responsible for handling its part of data and no one server is burdened. These servers are also called **shards**
- Every shard is an independent database. All the shards collectively make up a single logical database
- Sharding reduces the operations count handled by each shard. For example, when data is inserted, only the shards responsible for storing those records need to be accessed.
- The processes that need to be handled by each shard reduce as the cluster grows because the subset of data that the shard holds reduces. This leads to an increase in the throughput and capacity horizontally.
- Let's assume you have a database that is **1TB** in size. If the number of shards is 4, you will have approximately 265GB of data handled by each shard, whereas if the number of shards is increased to 40, only 25GB of data will be held on each shard.

Sharding in MongoDB



Sharding in MongoDB

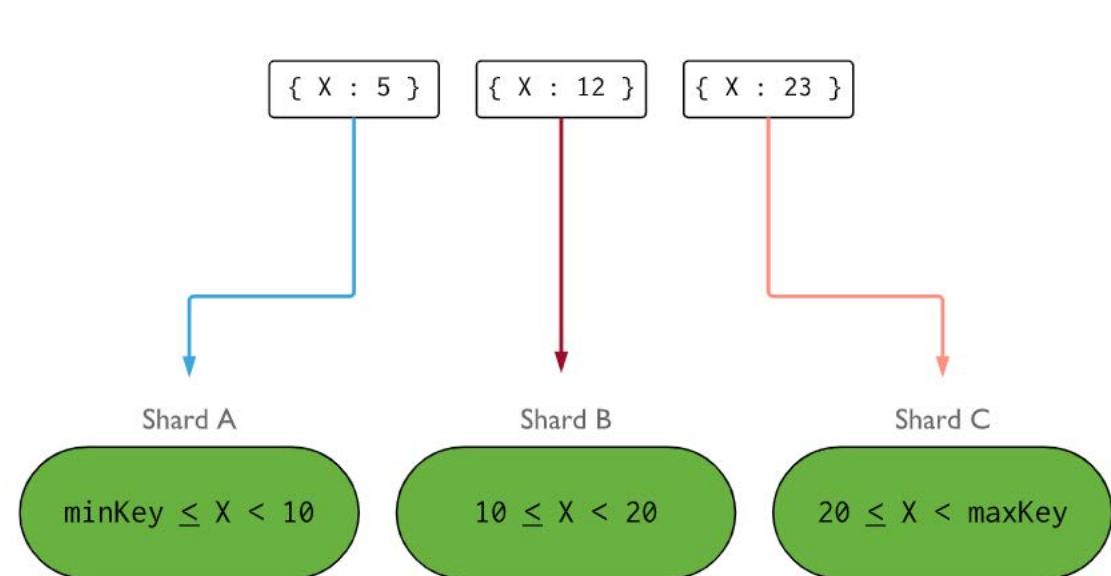
- Sharding increases deployment complexity
- Use sharding in the following instances:
 - The size of the dataset is huge and it has started challenging the capacity of a single system.
 - Since memory is used by MongoDB for quickly fetching data, it becomes important to scale out when the active work set limits are set to reach.
 - If the application is write-intensive, sharding can be used to spread the writes across multiple servers.
- Sharding is enabled in MongoDB via sharded clusters. The following are the components of a sharded cluster:
 - Shards
 - mongos
 - Config servers
- The shard is the component where the actual data is stored. For the sharded cluster, it holds a subset of data and can either be a mongod or a replica set. All shard's data combined together forms the complete dataset for the sharded cluster.
- Sharding is enabled per collection basis, so there might be collections that are not sharded. In every sharded cluster there's a primary shard where all the unsharded collections are placed in addition to the sharded collection data.

Sharding in MongoDB

- Config servers hold the sharded cluster's metadata. This metadata depicts the sharded system state and organization.
- The config server stores data for a single sharded cluster. The config servers should be available for the proper functioning of the cluster.
- One config server can lead to a cluster's single point of failure. For production deployment it's recommended to have at least three config servers, so that the cluster keeps functioning even if one config server is not accessible.
- A config server stores the data in the config database, which enables routing of the client requests to the respective data. This database should not be updated.
- MongoDB writes data to the config server only when the data distribution has changed for balancing the cluster. The **mongos** act as the routers. They are responsible for routing the read and write request from the application to the shards

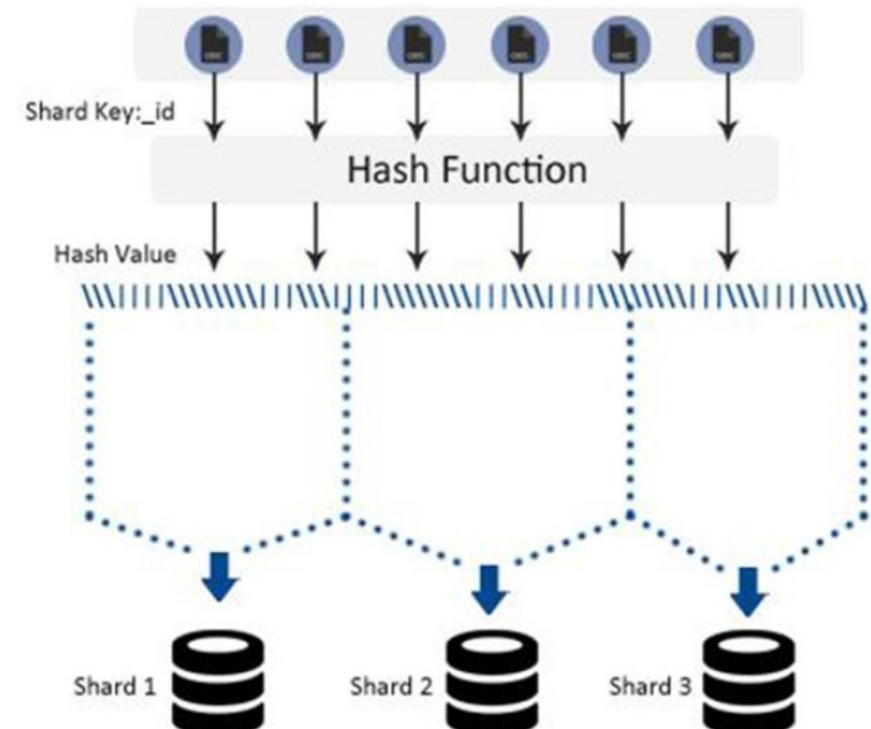
Types of Sharding – Range based sharding

- Any indexed single/compound field that exists within all documents of the collection can be a **shard key**.
- MongoDB divides the documents based on the value of the field into chunks and distributes them across the shards
- In range-based partitioning , the shard key values are divided into ranges. The following image illustrates a sharded cluster using the field X as the shard key. If the values for X have a large range, low frequency, and change at a non-monotonic rate,



Types of Sharding – Hash based sharding

- In hash-based partitioning , the data is distributed on the basis of the hash value of the shard field. If selected, this will lead to a more random distribution compared to range-based partitioning.
- It's unlikely that the documents with close shard key will be part of the same chunk. For example, for ranges based on the hash of the id field, there will be a straight line of hash values, which will again be partitioned on basis of the number of shards. On the basis of the hash values, the documents will lie in either of the shards.
- Hashed keys are ideal for shard keys with fields that change monotonically like ObjectId values or timestamps. A good example of this is the default `_id` field, assuming it only contains ObjectId values.



Chunks

- The data is moved between the shards in form of chunks. The shard key range is further partitioned into subranges, which are also termed as **chunks**
- For a sharded cluster, 64MB is the default chunk size. In most situations, this is an apt size for chunk slitting and migration.
- Let's discuss the execution of sharding and chunks with an example. Say you have a blog posts collection which is sharded on the field date . This implies that the collection will be split up on the basis of the date field values. Let's assume further that you have three shards. In this scenario the data might be distributed across shards as follows:
 - Shard #1: Beginning of time up to July 2009
 - Shard #2: August 2009 to December 2009
 - Shard #3: January 2010 to through the end of time
 - In order to retrieve documents from January 1, 2010 until today, the query is sent to mongos. In this scenario,
 1. The client queries mongos.
 2. The mongos know which shards have the data, so mongos sends the queries to Shard #3.
 3. Shard #3 executes the query and returns the results to mongos.
 4. Mongos combines the data received from various shards, which in this case is Shard #3 only, and returns the final result back to the client.
- Let's consider another scenario where you insert a new document. The new document has today's date.
- The sequences of events are as follows:
 1. The document is sent to the mongos.
 2. Mongos checks the date and on basis of that, sends the document to Shard #3.
 3. Shard #3 inserts the document.

AWS DynamoDB

- Serverless
- Key value + Document model
- Availability, durability, and fault tolerance are built-in
- ACID transactions
- Active-active replication with global tables
- Amazon DynamoDB Streams as part of an event-driven architecture
- Secondary indexes
- Fine grained access control
- Encryption at rest
- Point-in-time recovery
- On-demand backup and restore
- Read/write capacity modes
- Standard Infrequent Access (Standard-IA) table class