O'REILLY®

Compliments of Lightbend

Fast Data Architectures for Streaming Applications

Getting Answers Now from Data Sets That Never End

Dean Wampler, PhD

SECOND EDITION

Fast Data Architectures for Streaming Applications

Getting Answers Now from Data Sets That Never End

Dean Wampler, PhD

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Fast Data Architectures for Streaming Applications
by Dean Wampler

Printed in the United States of America.

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

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

Acquisitions Editor: Jonathan Hassell
Production Editor: Justin Billing
Copyeditor: Rachel Monaghan
Proofreader: James Fraleigh

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2016: First Edition
October 2018: Second Edition

Revision History for the Second Edition
2018-10-15: First Release

This work is part of a collaboration between O'Reilly and Lightbend. See our *statement of editorial independence*.

978-1-492-04679-0

[LSI]

Table of Contents

CHAPTER 1
Introduction

Until recently, *big data* systems have been batch oriented, where data is captured in distributed filesystems or databases and then processed in batches or studied interactively, as in data warehousing scenarios. Now, it is a competitive disadvantage to rely exclusively on batch-mode processing, where data arrives without immediate extraction of valuable information.

Hence, big data systems are evolving to be more stream oriented, where data is processed as it arrives, leading to so-called *fast data* systems that ingest and process continuous, potentially infinite data streams.

Ideally, such systems still support batch-mode and interactive processing, because traditional uses, such as data warehousing, haven't gone away. In many cases, we can rework batch-mode analytics to use the same streaming infrastructure, where we treat our batch data sets as finite streams.

This is an example of another general trend, the desire to reduce operational overhead and maximize resource utilization across the organization by replacing lots of small, special-purpose clusters with a few large, general-purpose clusters, managed using systems like Kubernetes or Mesos. While isolation of some systems and workloads is still desirable for performance or security reasons, most applications and development teams benefit from the ecosystems around larger clusters, such as centralized logging and monitoring, universal CI/CD (continuous integration/continuous delivery) pipe-

lines, and the option to scale the applications up and down on demand.

In this report, I'll make the following core points:

- Fast data architectures need a stream-oriented data backplane for capturing incoming data and serving it to consumers. Today, Kafka is the most popular choice for this backplane, but alternatives exist, too.
- Stream processing applications are "always on," which means they require greater resiliency, availability, and dynamic scalability than their batch-oriented predecessors. The *microservices* community has developed techniques for meeting these requirements. Hence, streaming systems need to look more like microservices.
- If we extract and exploit information more quickly, we need a more integrated environment between our microservices and stream processors, requiring fast data architectures that are flexible enough to support heterogeneous workloads. This requirement dovetails with the trend toward large, heterogeneous clusters.

I'll finish this chapter with a review of the history of big data and batch processing, especially the classic Hadoop architecture for big data. In subsequent chapters, I'll discuss how the changing landscape has fueled the emergence of stream-oriented, fast data architectures and explore a representative example architecture. I'll describe the requirements these architectures must support and the characteristics of specific tools available today. I'll finish the report with a look at an example IoT (Internet of Things) application that leverages machine learning.

A Brief History of Big Data

The emergence of the internet in the mid-1990s induced the creation of data sets of unprecedented size. Existing tools were neither scalable enough for these data sets nor cost-effective, forcing the creation of new tools and techniques. The "always on" nature of the internet also raised the bar for availability and reliability. The big data ecosystem emerged in response to these pressures.

At its core, a big data architecture requires three components:

Storage
: A scalable and available storage mechanism, such as a distributed filesystem or database

Compute
: A distributed compute engine for processing and querying the data at scale

Control plane
: Tools for managing system resources and services

Other components layer on top of this core. Big data systems come in two general forms: databases, especially the NoSQL variety, that integrate and encapsulate these components into a database system, and more general environments like Hadoop, where these components are more exposed, providing greater flexibility, with the trade-off of requiring more effort to use and administer.

In 2007, the now-famous Dynamo paper accelerated interest in NoSQL databases, leading to a "Cambrian explosion" of databases that offered a wide variety of persistence models, such as document storage (XML or JSON), key/value storage, and others. The CAP theorem emerged as a way of understanding the trade-offs between data consistency and availability guarantees in distributed systems when a network partition occurs. For the always-on internet, it often made sense to accept *eventual consistency* in exchange for greater availability. As in the original Cambrian explosion of life, many of these NoSQL databases have fallen by the wayside, leaving behind a small number of databases now in widespread use.

In recent years, SQL as a query language has made a comeback as people have reacquainted themselves with its benefits, including conciseness, widespread familiarity, and the performance of mature query optimization techniques.

But SQL can't do everything. For many tasks, such as data cleansing during ETL (extract, transform, and load) processes and complex event processing, a more flexible model was needed. Also, not all data fits a well-defined schema. Hadoop emerged as the most popular open-source suite of tools for general-purpose data processing at scale.

Why did we start with batch-mode systems instead of streaming systems? I think you'll see as we go that streaming systems are much harder to build. When the internet's pioneers were struggling to gain control of their ballooning data sets, building batch-mode architectures was the easiest problem to solve, and it served us well for a long time.

Batch-Mode Architecture

Figure 1-1 illustrates the "classic" Hadoop architecture for batch-mode analytics and data warehousing, focusing on the aspects that are important for our discussion.

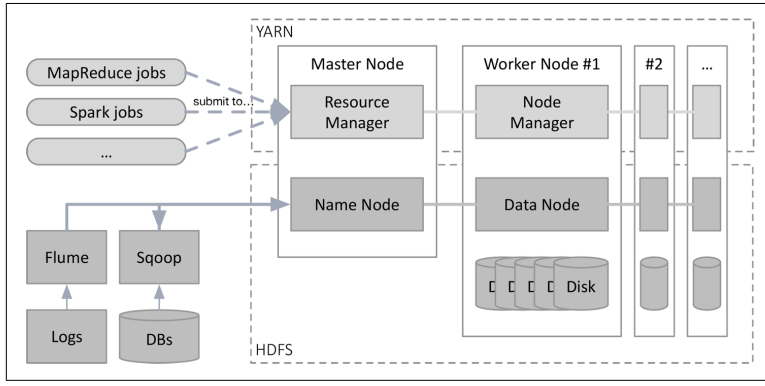

Figure 1-1. Classic Hadoop architecture

In this figure, logical subsystem boundaries are indicated by dashed rectangles. They are clusters that span physical machines, although HDFS and YARN (Yet Another Resource Negotiator) services share the same machines to benefit from data locality when jobs run.

Data is ingested into the persistence tier, into one or more of the following: HDFS (Hadoop Distributed File System), AWS S3, SQL and NoSQL databases, search engines like Elasticsearch, and other systems. Usually this is done using special-purpose services such as Flume for log aggregation and Sqoop for interoperating with databases.

Later, analysis jobs written in Hadoop MapReduce, Spark, or other tools are submitted to the Resource Manager for YARN, which decomposes each job into tasks that are run on the worker nodes, managed by Node Managers. Even for interactive tools like Hive

and Spark SQL, the same job submission process is used when the actual queries are executed as jobs.

Table 1-1 gives an idea of the capabilities of such batch-mode systems.

Table 1-1. Batch-mode systems

Metric	Sizes and units
Data sizes per job	TB to PB
Time between data arrival and processing	Minutes to hours
Job execution times	Seconds to hours

So, the newly arrived data waits in the persistence tier until the next batch job starts to process it.

In a way, Hadoop is a database *deconstructed*, where we have explicit separation between storage, compute, and management of resources and compute processes. In a regular database, these subsystems are hidden inside the "black box." The separation gives us more flexibility and reduces cost, but requires us to do more work for administration.

CHAPTER 2
The Emergence of Streaming

Fast-forward to the last few years. Now imagine a scenario where Google still relies on batch processing to update its search index. Web crawlers constantly provide data on web page content, but the search index is only updated every hour, let's say.

Suppose a major news story breaks and someone does a Google search for information about it, assuming they will find the latest updates on a news website. They will find nothing if it takes up to an hour for the next update to the index that reflects these changes. Meanwhile, suppose that Microsoft Bing does incremental updates to its search index as changes arrive, so Bing can serve results for breaking news searches. Obviously, Google is at a big disadvantage.

I like this example because indexing a corpus of documents can be implemented very efficiently and effectively with batch-mode processing, but a streaming approach offers the competitive advantage of timeliness. Couple this scenario with problems that are more obviously "real time," like location-aware mobile apps and detecting fraudulent financial activity as it happens, and you can see why streaming is so hot right now.

However, streaming imposes significant new operational challenges that go far beyond just making batch systems run faster or more frequently. While batch jobs might run for hours, streaming jobs might run for weeks, months, even years. Rare events like network partitions, hardware failures, and data spikes become inevitable if you run long enough. Hence, streaming systems have increased operational complexity compared to batch systems.

Streaming also introduces new semantics for analytics. A big surprise for me is how SQL, the quintessential tool for batch-mode analysis and interact exploration, has emerged as a popular language for streaming applications, too, because it is concise and easier to use for nonprogrammers. *Streaming SQL* systems rely on *windowing*, usually over ranges of time, to enable operations like `JOIN` and `GROUP BY` to be usable when the data set is never-ending.

For example, suppose I'm analyzing customer activity as a function of location, using zip codes. I might write a classic `GROUP BY` query to count the number of purchases, like the following:

```
SELECT zip_code, COUNT(*) FROM purchases GROUP BY zip_code;
```

This query assumes I have all the data, but in an infinite stream, I never will, so I can never stop waiting for all the records to arrive. Of course, I could always add a `WHERE` clause that looks at yesterday's data, for example, but when can I be sure that I've received all of the data for yesterday, or for any time window I care about? What about a network outage that delays reception of data for hours?

Hence, one of the challenges of streaming is knowing when we can reasonably assume we have all the data for a given context. We have to balance this desire for correctness against the need to extract insights as quickly as possible. One possibility is to do the calculation when I need it, but have a policy for handling late arrival of data. For some applications, I might be able to ignore the late arrivals, while for other applications, I'll need a way to update previously computed results.

Streaming Architecture

Because there are so many streaming systems and ways of doing streaming, and because everything is evolving quickly, we have to narrow our focus to a representative sample of current systems and a reference architecture that covers the essential features.

Figure 2-1 shows this fast data architecture.

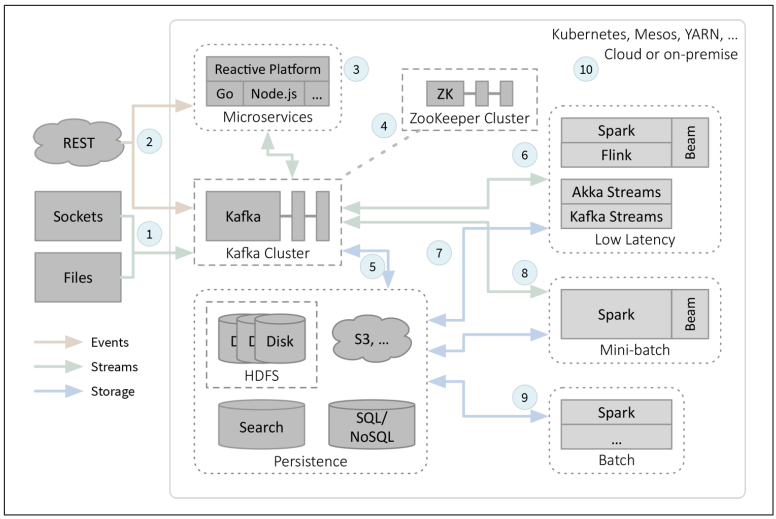

Figure 2-1. Fast data (streaming) architecture

There are more parts in Figure 2-1 than in Figure 1-1, so I've numbered elements of the figure to aid in the discussion that follows. Mini-clusters for Kafka, ZooKeeper, and HDFS are indicated by dashed rectangles. General functional areas, such as persistence and low-latency streaming engines, are indicated by the dotted, rounded rectangles.

Let's walk through the architecture. Subsequent sections will explore the details:

1. Streams of data arrive into the system from several possible sources. Sometimes data is read from files, like logs, and other times data arrives over sockets from servers within the environment or from external sources, such as telemetry feeds from IoT devices in the field, or social network feeds like the Twitter "firehose." These streams are typically *records*, which don't require individual handling like *events* that trigger state changes. They are ingested into a distributed Kafka cluster for scalable, durable, reliable, but usually temporary, storage. The data is organized into *topics*, which support multiple producers and consumers per topic and some ordering guarantees. Kafka is the backbone of the architecture. The Kafka cluster may use dedicated servers, which provides maximum load scalability and minimizes the risk of compromised performance due to "noisy neighbor" services misbehaving on the same machines.

On the other hand, strategic colocation of some other services can eliminate network overhead. In fact, this is how Kafka Streams works, as a library on top of Kafka (see also number 6).

2. REST (Representational State Transfer) requests are often synchronous, meaning a completed response is expected "now," but they can also be asynchronous, where a minimal acknowledgment is returned now and the completed response is returned later, using WebSockets or another mechanism. Normally REST is used for sending *events* to trigger state changes during sessions between clients and servers, in contrast to *records* of data. The overhead of REST means it is less ideal as a data ingestion channel for high-bandwidth data flows. Still, REST for data ingestion into Kafka is still possible using custom microservices or through Kafka Connect's REST interface.
3. A real environment will need a family of microservices for management and monitoring tasks, where REST is often used. They can be implemented with a wide variety of tools. Shown here are the Lightbend Reactive Platform (RP), which includes Akka, Play, Lagom, and other tools, and the Go and Node.js ecosystems, as examples of popular, modern tools for implementing custom microservices. They might stream state updates to and from Kafka, which is also a good way to integrate our time-sensitive analytics with the rest of our microservices. Hence, our architecture needs to handle a wide range of application types and characteristics.
4. Kafka is a distributed system and it uses ZooKeeper (ZK) for tasks requiring consensus, such as leader election and storage of some state information. Other components in the environment might also use ZooKeeper for similar purposes. ZooKeeper is deployed as a cluster, often with its own dedicated hardware, for the same reasons that Kafka is often deployed this way.
5. With Kafka Connect, raw data can be persisted from Kafka to longer-term, persistent storage. The arrow is two-way because data from long-term storage can also be ingested into Kafka to provide a uniform way to feed downstream analytics with data. When choosing between a database or a filesystem, keep in mind that a database is best when row-level access (e.g., CRUD operations) is required. NoSQL provides more flexible storage and query options, consistency versus availability (CAP) tradeoffs, generally better scalability, and often lower operating costs,

while SQL databases provide richer query semantics, especially for data warehousing scenarios, and stronger consistency. A distributed filesystem, such as HDFS, or object store, such as AWS S3, offers lower cost per gigabyte storage compared to databases and more flexibility for data formats, but is best used when scans are the dominant access pattern, rather than per-record CRUD operations. A search appliance, like Elasticsearch, is often used to index data for fast queries.

6. For low-latency stream processing, the most robust mechanism is to ingest data from Kafka into the stream processing engine. There are quite a few engines currently vying for attention, and I'll discuss four widely used engines that cover a spectrum of needs.[1]

   You can evaluate other alternatives using the concepts we'll discuss in this report. Apache Spark's Structured Streaming and Apache Flink are grouped together because they run as distributed services to which you submit jobs to run. They provide similar, very rich analytics, inspired in part by Apache Beam, which has been a leader in defining advanced streaming semantics. In fact, both Spark and Flink can function as "runners" for data flows defined with Beam. Akka Streams and Kafka Streams are grouped together because they run as libraries that you embed in your microservices, providing greater flexibility in how you integrate analytics with other processing, with very low latency and lower overhead than Spark and Flink. Kafka Streams also offers a SQL query service, while Akka Streams integrates with the rich Akka ecosystem of microservice tools. Neither is designed to be as full-featured as Beam-compatible systems. All these tools support distribution in one way or another across a cluster (not shown), usually in collaboration with the underlying clustering system (e.g., Kubernetes, Mesos, or YARN; see number 10). It's unlikely you would need or want all four streaming engines. Results from any of these tools can be written back to new Kafka topics for downstream consumption. While it's possible to read and write data directly between

1 For a comprehensive list of Apache-based streaming projects, see Ian Hellström's article, "An Overview of Apache Streaming Technologies". Since this post and the first edition of my report were published, some of these projects have faded away and new ones have been created!

other sources and these tools, the durability and reliability of Kafka ingestion and the benefits of having one access method make it an excellent default choice despite the modest extra overhead of going through Kafka. For example, if a process fails, the data can be reread from Kafka by a restarted process. It is often not an option to requery an incoming data source directly.

7. Stream processing results can also be written to persistent storage and data can be ingested from storage. This is useful when *O(1)* access for particular records is desirable, rather than *O(N)* to scan a Kafka topic. It's also more suitable for longer-term storage than storing in a Kafka topic. Reading from storage enables analytics that combine long-term historical data and streaming data.
8. The *mini-batch* model of Spark, called Spark Streaming, is the original way that Spark supported streaming, where data is captured in fixed time intervals, then processed as a "mini batch." The drawback is longer latencies are required (100 milliseconds or longer for the intervals), but when low latency isn't required, the extra window of time is valuable for more expensive calculations, such as training machine learning models using Spark's MLlib or other libraries. As before, data can be moved to and from Kafka. However, Spark Streaming is becoming obsolete now that Structured Streaming is mature, so consider using the latter instead.
9. Since you have Spark and a persistent store, like HDFS or a database, you can still do batch-mode processing and interactive analytics. Hence, the architecture is flexible enough to support traditional analysis scenarios too. Batch jobs are less likely to use Kafka as a source or sink for data, so this pathway is not shown.
10. All of the above can be deployed in cloud environments like AWS, Google Cloud Environment, and Microsoft Azure, as well as on-premise. Cluster resources and job management can be managed by Kubernetes, Mesos, and Hadoop/YARN. YARN is most mature, but Kubernetes and Mesos offer much greater flexibility for the heterogeneous nature of this architecture.

When I discussed Hadoop, I mentioned that the three essential components are HDFS for storage, MapReduce and Spark for compute,

and YARN for the control plane. In the fast data architecture for streaming applications, the analogs are the following:

- Storage: Kafka
- Compute: Spark, Flink, Akka Streams, and Kafka Streams
- Control plane: Kerberos and Mesos, or YARN with limitations

Let's see where the sweet spots are for streaming jobs compared to batch jobs (Table 2-1).

Table 2-1. Streaming numbers for batch-mode systems

Metric	Sizes and units: Batch	Sizes and units: Streaming
Data sizes per job	TB to PB	MB to TB (in flight data)
Time between data arrival and processing	Seconds to hours	Microseconds to minutes
Job execution times	Minutes to hours	Microseconds to minutes

While the fast data architecture can store the same petabyte data sets, a streaming job will typically operate on megabyte to terabyte at any one time. A terabyte per minute, for example, would be a huge volume of data! The low-latency engines in Figure 2-1 operate at subsecond latencies, in some cases down to microseconds.

However, you'll notice that the essential components of a big data architecture like Hadoop are also present, such as Spark and HDFS. In large clusters, you can run your new streaming workloads and microservices, along with the batch and interactive workloads for which Hadoop is well suited. They are still supported in the fast data architecture, although the wealth of third-party add-ons in the Hadoop ecosystem isn't yet matched in the newer Kubernetes and Mesos communities.

What About the Lambda Architecture?

In 2011, Nathan Marz introduced the lambda architecture, a hybrid model that uses three *layers*:

- A *batch layer* for large-scale analytics over historical data
- A *speed layer* for low-latency processing of newly arrived data (often with approximate results)

- A *serving layer* to provide a query/view capability that unifies the results of the batch and speed layers.

The fast data architecture can be used to implement applications following the lambda architecture pattern, but this pattern has drawbacks.[2]

First, without a tool like Spark that can be used to implement logic that runs in both batch and streaming jobs, you find yourself implementing logic twice: once using the tools for the batch layer and again using the tools for the speed layer. The serving layer typically requires custom tools as well, to integrate the two sources of data. However, if everything is considered a "stream"—either finite, as in batch processing, or unbounded—then batch processing becomes just a subset of stream processing, requiring only a single implementation.

Second, the lambda architecture emerged before we understood how to perform the same accurate calculations in a streaming context that we were accustomed to doing in a batch context. The assumption was that streaming calculations could only be approximate, meaning that batch calculations would always be required for definitive results. That's changed, as we'll explore in Chapter 4.

In retrospect, the lambda architecture is an important transitional step toward the fast data architecture, although it can still be a useful pattern in some situations.

Now that we've completed our high-level overview, let's explore the core principles required for the fast data architecture, beginning with the need for a data backplane.

2 See Jay Kreps's Radar post, "Questioning the Lambda Architecture".

CHAPTER 3

Logs and Message Queues

"Everything is a file" is the powerful, unifying abstraction at the heart of *nix systems. It's proved surprisingly flexible and effective as a metaphor for over 40 years. In a similar way, "everything is a log" is the powerful abstraction for streaming architectures.

Message queues provide ideal semantics for managing producers that write messages to queues and consumers that read them, thereby joining subsystems together with a level of indirection that provides decoupling. Implementations can provide durable message storage with tunable persistence characteristics and other benefits.

Let's explore these two concepts, how they are different, their relative strengths and weaknesses, and a merger that provides the best of both worlds.

The Log Is the Core Abstraction

Logs have been used for a long time as a mechanism for services to output information about what they are doing, including implementation details and problems encountered, as well as application state transitions. Log entries may include a timestamp, a notion of "urgency" (e.g., error, warning, or informational), information about the process and/or machine, and an ad hoc text message with more details. The log entries may be written in space-separated text, JSON, or a binary format (useful for efficient transport and storage). Well-structured log entries at appropriate execution points are proxies for system events. The order of entries is significant, as it indi-

cates event sequencing and state transitions. While we often associate logs with files, this is just one possible storage mechanism.

The metaphor of a log generalizes to a wide class of data streams, such as these examples:

Service logs
: These are the logs that services write to capture implementation details as processing unfolds, especially when problems arise. These details may be invisible to users and not directly associated with the application's logical state.

Write-ahead logs for database CRUD transactions
: Each insert, update, and delete that changes state is an event. Many databases use a WAL (write-ahead log) internally to append such events durably and quickly to a filesystem before acknowledging the change to clients, after which time in-memory data structures and other, more permanent files are updated with the current state of the records. That way, if the database crashes after the WAL write completes, the WAL can be used to reconstruct and complete any in-flight transactions, once the database is running again.

Other state transitions
: User web sessions and automated processes, such as manufacturing and chemical processing, are examples of systems that routinely transition from one state to another. Logs are a popular way to capture and propagate these state transitions so that downstream consumers can process them as they see fit.

Telemetry from IoT devices
: Many widely deployed devices, including cars, phones, network routers, computers, airplane engines, medical devices, home automation devices, and kitchen appliances, are now capable of sending telemetry back to the manufacturer for analysis. Some of these devices also use remote services to implement their functionality, like location-aware and voice-recognition applications. Manufacturers use the telemetry to better understand how their products are used; to ensure compliance with licenses, laws, and regulations (e.g., obeying road speed limits); and for *predictive maintenance*, where anomalous behavior is modeled and detected that may indicate pending failures, so that proactive action can prevent service disruption.

Clickstreams

How do users interact with an application? Are there sections that are confusing or slow? Is the process of purchasing goods and services as streamlined as possible? Which application version leads to more purchases, A or B? Logging user activity allows for clickstream analysis.

Logs also enable two general architecture patterns that are popular in the microservice world: *event sourcing* and *command-query responsibility segregation* (CQRS).

To understand event sourcing, consider the database write-ahead log. It is a record of all changes (events) that have occurred. This log can be replayed ("sourced") to reconstruct the state of the database at any point in time, even though the only state visible to queries in most databases is the latest snapshot in time. Hence, an event source provides the ability to replay history and can be used to reconstruct a lost database, to replicate one instance to additional copies, to apply new analytics, and more.

Incremental replication supports CQRS. Having a separate data store for writes ("commands") versus reads ("queries") enables each one to be tuned and scaled independently, according to its unique characteristics. For example, I might have a few, high-volume writers, but a large number of occasional readers. If the write database goes down, reading can continue, at least for a while. Similarly, if reading becomes unavailable, writes can continue. The trade-off is accepting eventually consistency, as the read data stores will lag behind the write data stores.[1]

Hence, an architecture with logs at the core is a flexible architecture for a wide spectrum of applications.

Message Queues and Integration

Traditional message queues are first-in, first-out (FIFO) data structures. The word "message" is used historically here; the data can be any kind of *record*, *event*, or the like. The ordering is often by time of arrival, similar to logs. Each message queue can represent a logical

1 Jay Kreps doesn't use the term CQRS, but he discusses the advantages and disadvantages in practical terms in his Radar post, "Why Local State Is a Fundamental Primitive in Stream Processing".

concept, allowing readers to focus on the messages they care about and not have to process all of the messages in the system. This also promotes scalability through parallelism, as the processing of each queue can be isolated in its own process or thread. Most implementations allow more than one writer to insert messages and more than one reader to extract them.

All this is a good way to organize and use logs, too. There's a crucial difference in the reading semantics of logs versus queues, which means that queues and logs are not equivalent constructs.

For most message queue implementations, when a message is read, it is also deleted from the queue; that is, the message is "popped." You might have multiple stateless readers for parallelism, such as a pool of workers, each of which pops a message, processes it, and then comes back for a new one, while the other workers are doing the same thing in parallel. However, having more than one reader means that none of them will see *all* the messages in the queue. That's a disadvantage when we have multiple readers where each one does something different. We'll return to this crucial point in a moment.

But first, let's discuss a few real-world considerations for queues. To ensure that all messages are processed *at least once*, the queue may wait for acknowledgment from a reader before deleting a message, but this means that policies and enforcement mechanisms are required to handle concurrency cases such as when a second reader tries to pop (read) a message before the queue has received the acknowledgment from the first reader. Should the same message be given to the second reader, effectively implementing *at least once* behavior? (See "At Most Once. At Least Once. Exactly Once." on page 19.) Or should the second reader be given the next message in the queue instead, while waiting for the acknowledgment for the first message? What happens if the acknowledgment for the first message is never received? How long should we wait? Presumably a timeout occurs and the first message must be made available again for a subsequent reader. But what happens if we want to process the messages in the queue's original FIFO order? In this case the readers will need to coordinate to ensure proper ordering. Ugh…

At Most Once. At Least Once. Exactly Once.

In a distributed system, there are many things that can go wrong when messages are passed between processes. What should we do if a message fails to arrive? How do we know it failed to arrive? There are three behaviors we can strive to achieve.

At most once ("fire and forget") means the message is sent, but the sender doesn't care if it's received or lost. This is fine if data loss is not a concern (e.g., when feeding a dashboard). With no guarantees of message delivery, there is no overhead to ensure message delivery! Hence, this is the easiest behavior to support, with optimal performance.

At least once means that retransmission of a message will occur until an acknowledgment is received. Since a delayed acknowledgment from the receiver could be in flight when the sender retransmits the message, the message may be received more than once. This is the most practical model when message loss is not acceptable (e.g., for bank transactions) but duplication must be handled by the receiver.

Exactly once is the "unicorn" of message sending. It means a message is received once and *only* once. It is never lost and never repeated. This is the ideal scenario, because it is the easiest to reason about when you are managing system state. It is also *impossible* to implement in the general case,[2] but it can be implemented for specific cases, at least to a high degree of reliability.[3]

Practically, you often use *at least once* delivery combined with logic where applying duplicate updates is *idempotent*; they cause no state changes. Or, deduplicate messages by including a unique identifier or incrementing index and discard those messages that have already been seen.

Combining Logs and Queues

An implicit goal with logs is that *all* readers should be able to see the entire log, not just a subset of it. This is crucial for *stateful* process-

2 See Tyler Treat's blog post, "You Cannot Have Exactly-Once Delivery".

3 You can always concoct a failure scenario where some data loss will occur.

ing of the log for different purposes. For example, for a given log, we might have one reader that updates a database write-ahead log, another that feeds a dashboard, and another that is used for training machine learning models, all of which need to see all entries. With a traditional message queue, we can only have one reader so it sees all messages, and it would have to support all these downstream processing scenarios.

Hence, a log system does *not* pop entries on reading. They may live forever or the system may provide some mechanism to delete old entries. The log system allows each reader to decide at which offset into the log reading should start, which supports reprocessing part of the log or restarting a failed process where it left off. The reader can then scan the entries, in order, at its own pace, up to the latest entry.

This means that the log system must track the current *offset* into the log for each reader. The log system may also support configurable at-most-once, at-least-once, or exactly-once semantics.

To get the benefits of message queues, the log system can support multiple logs, each working like a message queue where the entries focus on the same area of interest and typically have the same schema. This provides the same organizational benefits as classic message queues.

The Case for Apache Kafka

Apache Kafka implements the combined log and message queue features just described, providing the best of both models. The Kafka documentation describes it as "a distributed, partitioned, replicated commit log service."

Kafka was invented at LinkedIn, where it matured into a highly reliable system with impressive scalability.[4]

Hence, Kafka is ideally suited as the backbone of fast data architectures.

Kafka uses the following terms for the concepts we've described in this chapter. I'll use the term *record* from now on instead of *entries*,

4 In 2015, LinkedIn's Kafka infrastructure surpassed 1.1 *trillion* messages per *day*, and it's been growing since then.

which I used for logs, and *messages*, which I used for message queues.

Topic
: The analog of a message queue where records of the same "kind" (and usually the same schema) are written.

Partition
: A way of splitting a *topic* into smaller sections for greater parallelism and capacity. While the topic is a logical grouping of records, it can be partitioned randomly or by hashing a key. Note that record order is only guaranteed for a partition, *not* the whole topic when it has more than one partition. This is often sufficient, as in many cases we just need to preserve ordering for messages with the same key, all of which will get hashed to the same partition. Partitions can be replicated across a Kafka cluster for greater resiliency and availability. For durability, each partition is written to a disk file and a record is not considered committed until it has been written to this file.

Producer
: Kafka's term for a writer.

Consumer
: Kafka's term for a reader. It is usually ideal to have one reader *instance* per partition. See the Kafka documentation for additional details.

Consumer group
: A set of *consumers* that covers the partitions in a topic.

Kafka will delete blocks of records, oldest first, based either on a user-specified retention time (the time to live, or TTL, which defaults to seven days), a maximum number of bytes allowed in the topic (the default is unbounded), or both.

The normal case is for each consumer to walk through the partition records in order, but since the consumer controls which offset is read next, it can read the records in any order it wants. The consumer offsets are actually stored by Kafka itself, which makes it easier to restart a failed consumer where it left off.

A topic is a big buffer between producers and consumers. It effectively decouples them, providing many advantages. The big buffer means data loss is unlikely when there is one instance of a consumer

for a particular logical function and it crashes. Producers can keep writing data to the topic while a new, replacement consumer instance is started, picking up where the last one left off.

Decoupling means it's easy for the numbers of producers and consumers to vary independently, either for scalability or to integrate new application logic with the topic. This is much harder to do if producers and consumers have direct connections to each other, for example using sockets.

Finally, the producer and consumer APIs are simple and narrow. They expose a narrow abstraction that makes them easy to use and also effectively hides the implementation so that many scalability and resiliency features can be implemented behind the scenes. Having one universal way of connecting services like this is very appealing for architectural simplicity and developer productivity.

Alternatives to Kafka

You might have noticed in Chapter 2 that we showed five options for streaming engines and three for microservice frameworks, but only one log-oriented data backplane option, Kafka. In 1979, the only relational database in the world was Oracle, but of course many alternatives have come and gone since then. Similarly, Kafka is by far the most widely used system of its kind today, with a vibrant community and a bright future. Still, there are a few emerging alternatives you might consider, depending on your needs: Apache Pulsar, which originated at Yahoo! and is now developed by Streaml.io, and Pravega, developed by Dell EMC.

I don't have the space here to compare these systems in detail, but to provide motivation for your own investigation, I'll just mention two advantages of Pulsar compared to Kafka, at least as they exist today.

First, if you prefer a message queue system, one designed for big data loads, Pulsar is actually implemented as a queue system that also supports the log model.

Second, in Kafka, each partition is explicitly tied to one file on one physical disk, which means that the maximum possible partition size is bounded by the hard drive that stores it. This explicit mapping also complicates scaling a Kafka topic by splitting it into more partitions, because of the data movement to new files and possibly new disks that is required. It also makes scaling *down*, by consolidat-

ing partitions, sufficiently hard that it is almost never done. Because Pulsar treats a partition as an abstraction, decoupled from how the partition is actually stored, the Pulsar implementation is able to store partitions of unlimited size. Scaling up and down is much easier, too.

To be abundantly clear, I'm not arguing that Pulsar is *better* than Kafka. These two advantages may be of no real value to you, and there are many other pros and cons of these systems to consider.

When Should You Not Use a Log System?

Finally, all choices have disadvantages, including Kafka. Connecting two services through a Kafka topic has the disadvantages of extra overhead, including disk I/O to persist the log updates, and the latency between adding a record to the log and a consumer reading it, where there could be many other records ahead of your record in the log.

Put another way, sending a record from one service to another using a socket connection, such as REST, shared memory, or another IPC (interprocess communication) primitive will usually be faster and consume fewer system resources. You will give up all the advantages of Kafka, including decoupling of services and greater resiliency and flexibility.

So, use Kafka topics as the default choice, but if you have extremely tight latency requirements or lots of small services where messaging overhead would be a significant percentage of the overall compute time, consider which connections should happen without using Kafka. On the other hand, remember that *premature optimization is the root of all evil*.[5]

Now that we've made the case for a data backplane system like Kafka, let's explore our options for processing this data with various streaming engines.

5 Donald Knuth, "Structured Programming with Goto Statements," *Computing Surveys* 6, no. 4 (1974): 261–301 (but possibly a rephrasing of an earlier quote from C. A. R. Hoare).

CHAPTER 4

How Do You Analyze Infinite Data Sets?

Infinite data sets raise important questions about how to do certain operations when you don't have all the data and never will. In particular, what do classic SQL operations like `GROUP BY` and `JOIN` mean in this context? What about statistics like *min*, *max*, and *average*?

A theory of streaming semantics has emerged that provides the answer. Central to this theory is the idea that *aggregation* operations like these make sense only in the context of windows of data, often over a fixed range of time.

Apache Beam, an open source streaming engine based on Google Dataflow, is arguably the streaming engine with the most sophisticated formulation of these semantics. It has become the gold standard for defining how precise analytics should be performed in real-world streaming scenarios. Although Beam is not as widely used as the other streaming engines discussed in this report, the designs of these engines were strongly influenced by Beam.

To actually use Beam, a third-party "runner" is required to execute Beam data flows. In the open source world, this functionality has been implemented for Flink and Spark, while Google's own runner is its cloud service, Cloud Dataflow. This means you can write Beam data flows and run them with these other tools. Not all constructs defined by Beam are supported by all runners. The Beam documentation has a Capability Matrix that shows what features each runner

supports. There are even semantics defined that Beam and Google Cloud Dataflow themselves don't yet support!

Usually, when a runner supports a Beam construct, the runner also provides access to the feature in its "native" API. So, if you don't need runner portability, you might use that API instead.

For space reasons, I can only provide a sketch of these advanced semantics here, but I believe that every engineer working on streaming pipelines should take the time to understand them in depth. Tyler Akidau, the leader of the Beam/Dataflow team, has written two O'Reilly Radar blog posts and co-wrote an O'Reilly book explaining these details:

- Tyler Akidau, "The World Beyond Batch: Streaming 101", August 5, 2015, O'Reilly.
- Tyler Akidau, "The World Beyond Batch: Streaming 102", January 20, 2016, O'Reilly.
- Tyler Akidau, Slava Chernyak, and Reuven Lax, *Streaming Systems: The What, Where, When and How of Large-Scale Data Processing* (Sebastopol, CA: O'Reilly, 2018).

If you follow no other links in this report, at least read those two blog posts!

Streaming Semantics

Suppose we set out to build our own streaming engine. We might start by implementing two "modes" of processing, to cover a large fraction of possible scenarios: single-record processing and what I'll call "aggregation processing" over many records, including summary statistics and operations like `GROUP BY` and `JOIN` queries.

Single-record processing is the simplest case to support, where we process each record individually. We might trigger an alarm on an error record, or filter out records that aren't interesting, or transform each record into a more useful format. Lots of ETL (extract, transform, and load) scenarios fall into this category. All of these actions can be performed one record at a time, although for efficiency reasons we might do small batches of them at a time. This simple processing model is supported by all the low-latency tools introduced in Chapter 2 and discussed in more depth shortly.

The next level of sophistication is aggregations of records. Because the stream may be infinite, the simplest approach is to trigger a computation over a fixed window,[1] usually defined by a start and end time, in which case the actual number of records can vary from one window to the next. Another option is to use windows with a fixed number of records.

Time windows are more common, because they are usually tied to a business requirement, such as counting the number of transactions per minute. Suppose we implement this requirement and also a requirement to segregate the transactions by geographic region. We collect the data coming in and at the end of each minute window, we perform these tasks on the accumulated data, while the next minute's data is being accumulated. This fixed-window approach is the core construct in Spark's original Streaming model, based on *mini-batch* processing. It was a clever way to adapt Spark, which started life as a batch-mode tool, to provide a stream processing capability.

This notion of windows also applies to SQL extensions for streaming, which I mentioned in Chapter 2.

But the fixed-window model has challenges. Time is a first-class concern in streaming. We almost always want to know when things happened and the order in which they happened, especially if state transitions are involved.

There is a difference between *event time*, when something happened on a particular server, and *processing time*, some later time when the record that represents the event is processed, perhaps on a different server. Processing time is easiest to handle; for our example, we just trigger the computation once a minute on the processing server. However, event time is almost always what we *need*. Most likely, we need an accurate count of transactions per minute per country for accounting reasons. If we're tracking potential fraudulent activity, we may need accuracy for legal reasons. So, a fixed-window model can't work with just the processing time alone.

1 Sometimes the word *window* is used in a slightly different context in streaming engines and the word *interval* is used instead for the idea we're discussing here. For example, in Spark Streaming's *mini-batch* system, I define a fixed *mini-batch interval* to capture data, but I process a *window* of one or more intervals together at a time.

Another complication is the fact that times across servers will never be completely synchronized. The widely used Network Time Protocol for clock synchronization is accurate to a few milliseconds, at best. Precision Time Protocol is submicrosecond, but not as widely deployed. If I'm trying to reconstruct a sequence of events based on event times for activity that spans servers, I have to account for the inevitable clock skew between the servers. Still, it may be sufficient if we accept the timestamp assigned to an event by the server where it was recorded.

Another implication of the difference between event and processing time is the fact that I can't really know that my processing system has received all of the records for a particular window of event time. Arrival delays could be significant, due to network delays or partitions, servers that crash and then reboot, mobile devices that leave and then rejoin the network, and so on. Even under normal operations, data travel is not instantaneous between systems.

Because of inevitable latencies, however small, the actual records in the mini-batch will include stragglers from the previous window of event time. After a network partition is resolved, events from the other side of the partition could be significantly late, perhaps many windows of time late.

Figure 4-1 illustrates event versus processing times.

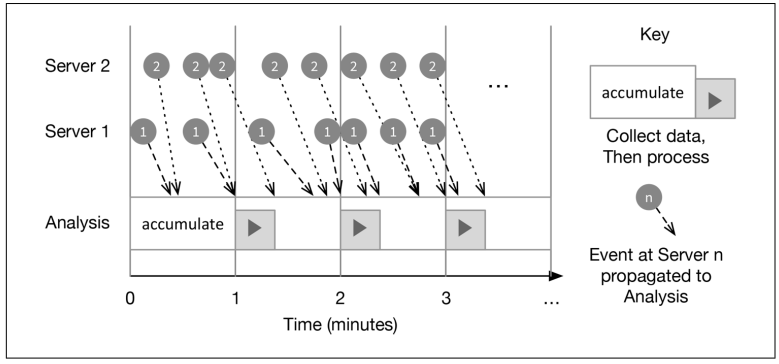

Figure 4-1. Event time versus processing time

Records generated on Servers 1 and 2 at particular event times may propagate to the Analysis server at different rates and traverse different distances. Most records will arrive in the same window in which they occurred, while others will arrive in subsequent win-

dows. During each minute, events that arrive are accumulated. At minute boundaries, the analysis task for the previous minute's events is invoked (represented by the right-facing arrowhead), but clearly *some data will always be missing*.

So, we really should delay processing until we have all the records for a given event-time window *or* decide when we can't wait any longer. This impacts correctness, too. For stream processing to replace batch-mode processing, we require the same correctness guarantees. Our transactions-per-minute calculation should have the same accuracy whether we do it incrementally over the incoming records or compute it tomorrow in a batch job that analyzes today's activity.

To balance these trade-offs, additional concepts are needed. We'll keep finite windows, but we'll add the concepts of *watermarks* and *triggers* for when processing should proceed. We also need to clarify how updates to previous results are handled:

Watermarks
: Used to indicate that all events relative to a window or other context have been received, so it's now safe to process the final result for the context.

Triggers
: A more general tool for invoking processing, including the detection of watermarks, a timer firing, or a threshold being reached. Previously, we just started processing when the window end time was reached. The results may be incomplete for a given context, if more data will arrive later.

Accumulation modes
: How should we treat multiple observations within the same window, whether they are disjoint or overlapping? If overlapping, do later observations override earlier ones or are they merged somehow?

It's great when we have a watermark that tells us we've seen everything for a context. If we haven't received one, we may need to trigger the calculation because downstream consumers can't wait any longer for the result, even if it's preliminary. We still need a plan for handling records that arrive too late.

One possibility is to structure our analysis so that everyone knows we're computing a preliminary result when a trigger occurs, then

accepts that an improved result will be forthcoming, if and when additional late records arrive. A dashboard might show a result that is approximate at first and grows increasingly correct over time. It's probably better to show *something* early on, as long as it's clear that the results aren't final.

However, suppose instead that these point-in-time analytics are written to yet another stream of data that feeds a billing system. Simply overwriting a previous value isn't an option. Accountants never modify an entry in bookkeeping. Instead, they add a second entry that corrects the earlier mistake. Similarly, our streaming tools need to support a way of correcting results, such as issuing *retractions*, followed by new values.

Recall that we had a second task to do: segregate transactions by geographic region. As long as our persistent storage supports incremental updates, this second task is less problematic; we just add late-arriving records as they arrive. Hence, scenarios that don't have an explicit time component are easier to handle.

We've just scratched the surface of the important considerations required for processing infinite streams in a timely fashion, while preserving correctness, robustness, and durability.

Now that you understand some of the important concepts required for effective stream processing, let's examine several of the currently available tools and how they support these concepts, as well as other potential requirements.

Which Streaming Engines Should You Use?

How do the streaming engines available today stand up? Which ones should you use?

There are five streaming engines shown in Figure 2-1. We discussed Beam already, because of its foundational role in defining the semantics of streaming. The other four cover a spectrum of features and ideal use cases. Each has its own vibrant community and ecosystem of other tools to leverage.

Criteria for Evaluating Streaming Engines

For a given application, the following criteria should be considered:

What is the latency budget?
: If you must process records within milliseconds, this will constrain your options. If your budget is minutes to hours, you have a lot of options; you could even run frequent, short batch jobs!

What is the volume of data to be processed per unit time?
: A streaming job may ultimately process petabytes of data, but if this happens one record at a time over a very long time, then almost anything can be made to work. If you're processing the Twitter firehose, then you'll need tools that provide lots of scalability through parallelism.

Which kinds of data processing will you do?
: Simple record transformations and filtering, like many ETL tasks, can be done with many tools. You'll need sophisticated tools if you are doing complex `GROUP BY` and `JOIN` operations over several streams and historical data. If you need the precise semantics discussed previously in this chapter, that will also constrain your choices.

How strong and dynamic is the community around the project?
: Is the project evolving quickly? Are there lots of people contributing enhancements and add-on libraries? Are quality standards high? Does it appear that the project will have a long and healthy life?

How do you build, deploy, and manage services?
: If your organization already has a mature CI/CD (continuous integration/continuous delivery) process for microservices, it would be very nice to continue using those tools for stream processing as well. If you are accustomed to running and using a diverse set of tools, then adding new ones will be less of a challenge.

Let's explore the four tools based on these criteria.

Spark and Flink: Scalable Data Processing Systems

Spark and Flink are examples of systems that automatically manage a group of processes across a cluster and partition and distribute your data to those processes. They handle high-volume scenarios very well and can also operate at relatively low latencies. They offer

rich options for stream processing, from simple transformations to subsets of the Beam semantics, either as data flows or SQL queries.

Both support different application deployment options and they are integrated with the popular resource managers: YARN, Kubernetes, and Mesos, which are used to obtain resources on demand. In the most common scenario, you submit a *job* to a master service, which decides how to partition the data set to be processed and then schedules *tasks* (JVM processes) to do the work of the job.

All of this work behind the scenes is done for you, as long as your job fits this runtime model. The APIs are reasonably intuitive and promote concise code. Today, they come in two basic forms:

Data flows
: A programming model based on sequencing transformations on collections. For Spark in particular, this API was heavily inspired by the Scala language collections API, which also inspired recent changes in Java's collections API. For problems that are naturally described as a sequence of processing steps, forming a DAG (directed acyclic graph), this approach is very flexible and powerful.

SQL
: Recall that SQL is a declarative language, where you specify the relational constraints and the query engine determines how to select and transform the data to satisfy those constraints. We saw earlier that SQL with extensions for streaming has become a popular choice for writing applications, because it is concise and easier to use by nonprogrammers.

Both models produce new streams, even SQL. In batch-mode SQL, the data *at rest* is processed to return a new, fixed data set. In the Streaming context, the input stream is transformed into an output stream matching the criteria of the query.

Both models require window primitives in order for them to work on infinite streams of data. Beam's concepts for windowing were influential in the designs of Flink and Spark.

Spark and Flink have different histories. Spark started as a batch-mode system, then added stream processing later. Flink started as a streaming engine first, with batch support added later.

The initial *Spark Streaming* system is based on mini-batch processing, with windows (called *intervals*) of approximately 100 milliseconds or more. Also, it only supports processing time windows. This API is based on the original storage model in Spark, RDDs (resilient, distributed data sets), which provides the previously mentioned data flow API inspired by the Scala collections API. This API comes in Scala, Java, Python, and R versions. Spark Streaming is now effectively deprecated, replaced by a new streaming system.

The new system, *Structured Streaming*, introduced in Spark version 2.0, is based on *SparkSQL*, Spark's declarative SQL system, with an idiomatic API in Scala, Java, Python, and R, as well as SQL itself. The declarative nature of SQL has provided the separation between user code and implementation that has allowed the Spark project to focus on aggressive internal optimizations, as well as support for advanced streaming semantics, like event-time processing. This is now the standard streaming API for Spark. A major goal of the Spark project is to promote so-called *continuous applications*, where the division between batch and stream processing is further reduced. In parallel, a new low-latency streaming engine has been developed to break through the 100-millisecond barrier of the original engine.

Spark also includes a machine learning library, *MLlib*, and many third-party libraries integrate with Spark.

Because Flink has always excelled at low-latency stream processing, it has maintained a lead over Spark in supporting advanced semantics. Specifically, it supports more of the semantics defined by Beam compared to Spark, as described in the Beam Capability Matrix. It also offers the more mature Beam runner of the two systems.

Flink also provides both a data-flow-style API in Scala and Java, as well as SQL. Integrated machine learning is less mature compared with Spark.

To conclude, Spark and Flink are comparable in the list of criteria we defined. Both provide rich options for processing, good low-latency and very good high-volume performance, and automatic data partitioning and task management. Both have vibrant and growing communities and tool ecosystems. Spark's community is much larger at this time, while Flink is more mature for stream processing. Both have very good quality.

Akka Streams and Kafka Streams: Data-Centric Microservices

Akka Streams and Kafka Streams are *libraries* that you integrate into applications, as opposed to standalone services like Spark and Flink. They are designed for the sweet spot of building microservices with integrated data processing, versus using separate services focused exclusively on data analytics. There are several advantages to this approach:

- You would rather use the same microservice-based development and production workflow for all services, versus supporting a more heterogeneous environment.
- You want more control over how applications are deployed and managed.
- You want more flexibility in how different domain logic is integrated together—for example, incorporating machine learning model training or serving into other microservices, rather than deploying them separately.

If you run separate data analytics (for example, Spark jobs), and separate microservices that use the results, you'll need to exchange the results between the services (for example, via a Kafka topic). This is a great pattern, of course, as I argued in Chapter 3, but also I pointed out that sometimes you don't want the extra overhead of going through Kafka. Nothing is faster with lower overhead than making a function call in the same process, which you can do if your data transformation is done by a library rather than a separate service! Also, a drawback of microservices is the increase in the number of different services you have to manage. You may want to minimize that number.

Because Akka Streams and Kafka Streams are libraries, the drawback of using them is that you have to handle some details yourself that Spark or Flink would do for you, like automatic task management and data partitioning. Now you really need to write all the logic for creating a runnable process and actually running it.

Despite these drawbacks, because Kafka Streams runs as a library on top of the Kafka cluster, it requires no additional cluster setup and modest additional management overhead to use.

You can use Akka Streams in a similar way, too, although you can leverage *Akka Cluster* for building highly distributed, resilient applications, when needed. The Akka ecosystem of which Akka Streams is a part—combined with the other Lightbend Reactive Platform projects, Play and Lagom—provides a full spectrum of tools and libraries for microservice development. While Kafka Streams is focused on reading and writing Kafka topics, the connector library *Alpakka* makes it relatively easy to connect different data sources and sinks to Akka Streams.

Both streaming libraries provide single-event processing with very low latency and high throughput. When exchanging data over a Kafka topic, you do need to carefully watch *consumer lag* (i.e., queue depth), which is a source of latency.

Kafka Streams doesn't attempt to provide a full spectrum of microservice tools, so it's common to embed Kafka Streams in applications built with other tools, like Lightbend Reactive Platform and the Spring Framework.

Akka Streams also implements the *Reactive Streams* specification. This is a simple, yet powerful, standard for defining composable streams with built-in *back pressure*, a flow control mechanism. If a single stream has built-in flow control, such that it can control how much data is pushed into it to avoid data loss, then this back pressure composes when these streams are joined together, either in the same process or across process boundaries. This keeps the internal stream "segments" robust against data loss and allows strategic decisions to be made at the entry points for the assembly, where it is more likely that a good strategy can be defined and implemented. Hence, reactive streams are a very robust mechanism for reliable processing. If you don't use a Kafka topic as a giant buffer between microservices, you *really need* a back pressure mechanism like this!

Neither library does automatic data partitioning and dynamic task management associated with these partitions, so they are not as well suited for high-volume scenarios as Spark and Flink. Nothing stops you from implementing this yourself, of course, but good luck…

When you are comparing these two libraries, perhaps the most important thing to understand is that each emerged out of a different community, which has influenced the features it offers.

It's useful to know that Kafka Streams was really designed with data processing in mind, while Akka Streams emerged out of the microservices focus of Akka. Compared to Flink and Spark, neither library is as advanced at the Beam-style streaming semantics, but Kafka Streams is very good at supporting many of the most common data processing scenarios, such as `GROUP BY` and `JOIN` operations and aggregations. When you want to see all records in a stream, the usual way of using Kafka topics, Kafka Streams offers a *KStream* abstraction. When you just want to see the latest value for a key—say, a statistic computed over data in a stream—then the *KTable* abstraction, analogous to a database table, makes this easy. There is even a SQL query capability that lets you query the state of a running Kafka Stream. In contrast to Spark and Flink, SQL is not offered as an API for writing applications.

Compared to Kafka Streams, Akka Streams is more microservice oriented and less data-analytics oriented. You can still do many operations like filtering and transforming, as well as aggregations like grouping and joining, but the latter are not as feature-complete as comparable SQL operations. Of the five systems we're discussing here, Akka Streams and Beam are the only ones that don't offer a SQL abstraction (although Beam's semantics fit the requirements for streaming SQL very well). Similarly, Akka Streams doesn't implement many of the Beam semantics that the other engines implement for handling late arrival of data, grouping by event time, and so on.

So why discuss Akka Streams then? Because it is unparalleled for defining complex event processing (CEP) scenarios, such as life cycle state transitions, alarm triggering of certain events, and building a wide range of microservices. Of the five engines we're discussing, only Akka Streams supports feedback loops in the data flow graph, so you're not restricted to DAGs. If you think of typical data processing as one end of a spectrum and general microservice capabilities as the other end, then Akka Streams sits closer to the general microservice end, while Kafka Streams sits closer to the data processing end.

Okay, So What Should I Use?

I hesitated including four streaming engines, plus Beam. I was concerned about the *paradox of choice*, where if you have too many choices, you might panic at the thought of making the wrong choice. The reason I included so many options is because a real environ-

ment may need several of them, and each one brings unique strengths, as well as weaknesses. There is certainly enough overlap that you won't need all of them and you shouldn't take on the burden of learning and using more than a few. Two will probably suffice —for example, one system like Spark or Flink and one microservice library like Akka Streams or Kafka Streams.

If you already use Spark for interactive SQL queries, machine learning, and batch jobs, then Spark Streaming is a good choice that covers a wide class of problems and lets you share logic between streaming and batch applications. To paraphrase an old saying about IBM, *no one gets fired for choosing Spark*.

If you need the sophisticated semantics and low latency provided by Beam, then use Flink with the Beam API or Flink's own API. Choose Flink if you really are focused on streaming and batch processing is not a major requirement.

Since we assume that Kafka is a given, use Kafka Streams for low-overhead processing that doesn't require the sophistication or flexibility provided by the other tools. The management overhead is minimal, once Kafka is running. ETL tasks like filtering, transformation, and many aggregation tasks are ideal for Kafka Streams. Use it for Kafka-centric microservices that process asynchronous data streams.

Are you coming to streaming data from a microservice background? Do you want one full-featured toolbox with the widest spectrum of options? Use Akka Streams to implement your general-purpose and data-processing microservices.

CHAPTER 5
Real-World Systems

Fast data architectures raise the bar for the "ilities" of distributed data processing. Whereas batch jobs seldom last more than a few hours, a streaming pipeline is designed to run for weeks, months, even years. If you wait long enough, even the most obscure problem is likely to happen.

The umbrella term *reactive systems* embodies the qualities that real-world systems must meet. These systems must be:

Responsive
: The system can always respond in a timely manner, even when it's necessary to respond that full service isn't available due to some failure.

Resilient
: The system is resilient against failure of any one component, such as server crashes, hard drive failures, or network partitions. Replication prevents data loss and enables a service to keep going using the remaining instances. Isolation prevents cascading failures.

Elastic
: You can expect the load to vary considerably over the lifetime of a service. Dynamic, automatic scalability, both up and down, allows you to handle heavy loads while avoiding underutilized resources in less busy times.

Message driven
: While fast data architectures are obviously focused on data, here we mean that all services respond to directed commands and queries. Furthermore, they use messages to send commands and queries to other services as well.

Classic big data systems, focused on batch and offline interactive workloads, have had less need to meet these qualities. Fast data architectures are just like other online systems where these qualities are necessary to avoid costly downtime and data loss. If you come from a big data engineering background, you are suddenly forced to learn new skills for distributed systems programming and operations.

Some Specific Recommendations

Most of the components we've discussed previously support the reactive qualities to one degree or another. Of course, you should follow all of the usual recommendations about good management and monitoring tools, disaster recovery plans, and so on, which I won't repeat here. That being said, here are some specific recommendations:

- Ingest all inbound data into Kafka first, then consume it with the stream processors and microservices. You get durable, scalable, resilient storage. You get support for multiple, decoupled consumers, replay capabilities, and the simplicity and power of event log semantics and topic organization as the backplane of your architecture.
- For the same reasons, write data back to Kafka for consumption by downstream services. Avoid direct connections between services, which are less resilient, unless latency concerns require direct connections.
- When using direct connections between microservices, use libraries that implement the Reactive Streams standard, for the resiliency provided by back pressure as a flow-control mechanism.
- Deploy to Kubernetes, Mesos, YARN, or a similar resource management infrastructure with proven scalability, resiliency, and flexibility. I don't recommend Spark's standalone-mode deployments, except for relatively simple deployments that

aren't mission critical, because Spark provides only limited support for these features.

- Choose your databases and other persistence stores wisely. Are they easy to manage? Do they provide distributed scalability? How resilient against data loss and service disruption are they when components fail? Understand the CAP trade-offs you need and how well they are supported by your databases. Should you really be using a relational database? I'm surprised how many people jump through hoops to implement transactions themselves because their NoSQL database doesn't provide them.
- Seek professional production support for your environment, even when using open source solutions. It's cheap insurance and it saves you time (which equals money).

CHAPTER 6
Example Application

Let's finish with a look at an example application, similar to systems that several Lightbend customers have implemented.[1] Here, telemetry for IoT (Internet of Things) devices is ingested into a central data center. Machine learning models are trained and served to detect anomalies, indicating that a hardware or software problem may be developing. If any are found, preemptive action is taken to avoid loss of service from the device.

Vendors of networking, storage, and medical devices often provide this service, for example.

Figure 6-1 sketches a fast data architecture implementing this system, adapted from Figure 2-1, with a few simplifications for clarity. As before, the numbers identify the diagram areas for the discussion that follows. The bidirectional arrows have two numbers, to discuss each direction separately.

1 You can find case studies at *lightbend.com/customers*.

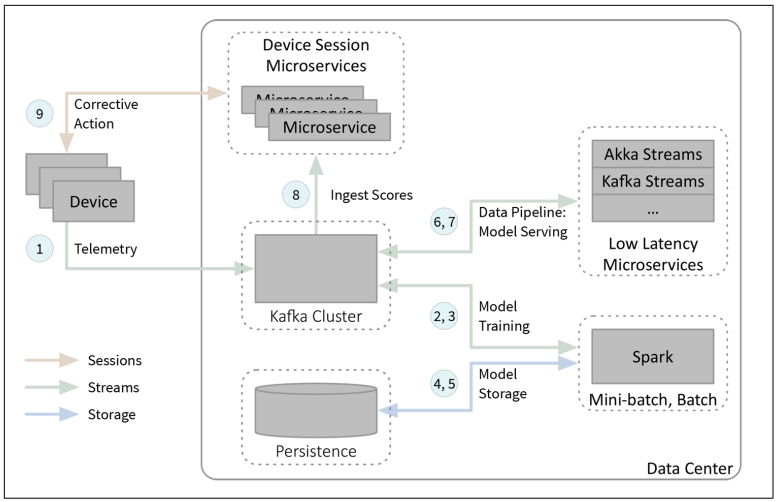

Figure 6-1. IoT anomaly detection example

There are three main segments of this diagram. After the telemetry is ingested (label 1), the first segment is for model training with periodic updates (labels 2 and 3), with access to persistent stores for saving models and reading historical data (labels 4 and 5). The second segment is for model serving—that is, scoring the telemetry with the latest model to detect potential anomalies (labels 6 and 7)—and the last segment is for handling detected anomalies (labels 8 and 9).

Let's look at the details of this figure:

1. Telemetry data from the devices in the field are streamed into Kafka, typically over asynchronous socket connections. The telemetry may include low-level machine and operating system metrics, such as component temperatures, CPU and memory utilization, and network and disk I/O performance statistics. Application-specific metrics may also be included, such as metrics for service requests, user interactions, state transitions, and so on. Various logs may be ingested, too. This data is captured into one or more Kafka topics.

2. The data is ingested into Spark for periodic retraining or updating of the anomaly detection model.[2] We use Spark because of its ability to work with large data sets (if we need to retrain using a lot of historical data), because of its integration with a variety of machine learning libraries, and because we only need to retrain occasionally, where hours, days, or even weeks is often frequently enough. Hence, this data flow does not have low-latency requirements, but may need to support processing a lot of data at once.
3. Updated model parameters are written to a new Kafka topic for downstream consumption by our separate serving system.
4. Updated model parameters are also written to persistent storage. One reason is to support auditing. Later on, we might need to know which version of the model was used to score a particular record. *Explainability* is one of the hard problems in neural networks right now; if our neural network rejects a loan application, we need to understand why it made that decision to ensure that bias against disadvantaged groups did not occur, for example.
5. The Spark job might read the last-trained model parameters from storage to make restarts faster after crashes or reboots. Any historical data needed for model training would also be read from storage.
6. There are two streams ingested from Kafka to the microservices used for streaming. The first is the original telemetry data that will be scored and the second is the occasional updates for the model parameters. Low-latency microservices are used for scoring when we have tight latency constraints, which may or may not be true for this anomaly detection scenerio, but would be true for fraud detection scenarios implemented in a similar way. Because we can score one record at a time, we don't need the same data capacity that model training requires. The extra flexibility of using microservices might also be useful.
7. In this example, it's not necessary to emit a new scored record for every input telemetry record; we only care about anomalous records. Hopefully, the output of such records will be very infre-

2 A real system might train and use several models for different purposes, but we'll just assume one model here for simplicity.

quent. So, we don't really need the scalability of Kafka to hold this output, but we'll still write these records to a Kafka topic to gain the benefits of decoupling from downstream consumers and the uniformity of doing all communications using one technique.

8. For the IoT systems we're describing, they may already have general microservices that manage sessions with the devices, used for handling requests for features, downloading and installing upgrades, and so on. We leverage these microservices to handle anomalies, too. They monitor the Kafka topic with anomaly records.
9. When a potential anomaly is reported, the microservice supporting the corresponding device will begin the recovery process. Suppose a hard drive appears to be failing. It can move data off the hard drive (if it's not already replicated), turn off the drive, and notify the customer's administrator to replace the hard drive when convenient.

The auditing requirement discussed for label 4 suggests that a version marker should be part of the model parameters used in scoring and it should be added to each record along with the score. An alternative might be to track the timestamp ranges for when a particular model version was used, but keep in mind our previous discussion about the difficulties of synchronizing clocks!

Akka Actors are particularly nice for implementing the "session" microservices. Because they are so lightweight, you can create one instance of a session actor per device. It holds the state of the device, mirrors state transitions, services requests, and the like, in parallel with and independent of other session actors. They scale very well for a large device network.

One interesting variation is to move model scoring down to the device itself. This approach is especially useful for very latency-sensitive scoring requirements and to mitigate the risk of having no scoring available when the device is not connected to the internet.

Figure 6-2 shows this variation.

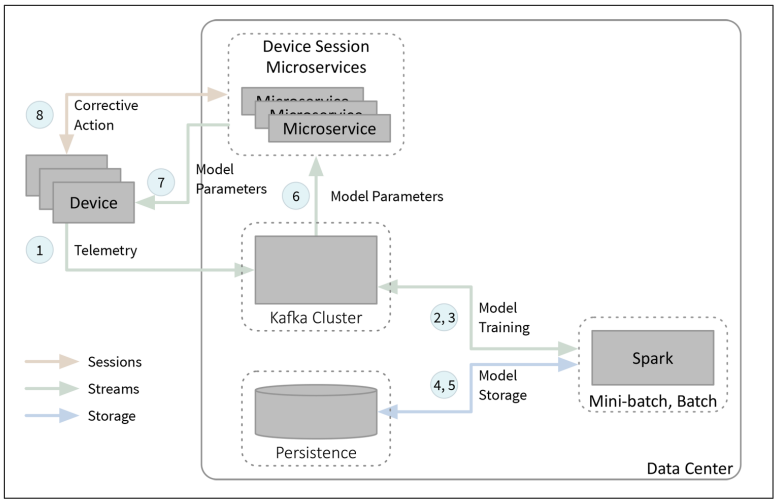

Figure 6-2. IoT anomaly detection scoring on the devices

The first five labels are unchanged.

6. Where previously anomaly records were ingested by the session microservices, now they read model parameter updates.
7. The session microservices push model parameter updates to the devices.
8. The devices score the telemetry locally, invoking corrective action when necessary.

Because training doesn't require low latency, this approach reduces the "urgency" of ingesting telemetry into the data center. It will reduce network bandwidth if the telemetry needed for training can be consolidated and sent in bursts.

Other Machine Learning Considerations

Besides anomaly detection, other ways we might use machine learning in these examples include the following:

Voice interface
: Interpret and respond to voice commands for service.

Improved user experience
: Study usage patterns to optimize difficult-to-use features.

Recommendations
: Recommend services based on usage patterns and interests.

Automatic tuning of the runtime environment
: In a very large network of devices and services, usage patterns can change dramatically over time and other changed circumstances. Usage spikes are common. Hence, being able to automatically tune how services are distributed and used, as well as how and when they interact with remote services, can make the user experience better and the overall system more robust.

Model drift or *concept drift* refers to how a model may become stale over time as the situation it models changes. For example, new ways of attempting fraud are constantly being invented. For some algorithms or systems, model updates will require retraining from scratch with a large historical data set. Other systems support incremental updates to the model using just the data that has been gathered since the last update. Fortunately, it's rare for model drift to occur quickly, so frequent retraining is seldom required.

We showed separate systems for model training and scoring, Spark versus Akka Streams. This can be implemented in several ways:

- For simple models, like logistic regression, use separate implementations, where parameters output by the training implementation are plugged into the serving system.
- Use the same machine learning library in both training and serving systems. Many libraries are agnostic to the runtime environment and can be linked into a variety of application frameworks.
- Run the machine learning environment as a separate service and request training and scoring through REST invocations or other means. Be careful about the overhead of REST calls in high-volume, low-latency scenarios.

CHAPTER 7
Recap and Where to Go from Here

Fast data is the natural evolution of big data to a stream-oriented workflow that allows for more rapid information extraction and exploitation, while still enabling classic batch-mode analytics, data warehousing, and interactive queries.

Long-running streaming jobs raise the bar for a fast data architecture's ability to stay resilient, scale up and down on demand, remain responsive, and be adaptable as tools and techniques evolve.

There are many tools with various levels of support for sophisticated stream processing semantics, other features, and deployment scenarios. I didn't discuss all the possible engines. I omitted those that appear to be declining in popularity, such as Storm and Samza, as well as newer but still obscure options. There are also many commercial tools that are worth considering. However, I chose to focus on the current open source choices that seem most important, along with their strengths and weaknesses.

I encourage you to explore the links to additional information throughout this report and in the next section. Form your own opinions and let me know what you discover and the choices you make. You can reach me through email, *dean.wampler@lightbend.com*, and on Twitter, @deanwampler.

At Lightbend, we've been working hard to build tools, techniques, and expertise to help our customers succeed with fast data. Please visit us at *lightbend.com/fast-data-platform* for more information.

Additional References

The following references, some of which were mentioned already in the report, are very good for further exploration:

- Tyler Akidau, "The World Beyond Batch: Streaming 101", August 5, 2015, O'Reilly.
- Tyler Akidau, "The World Beyond Batch: Streaming 102", January 20, 2016, O'Reilly.
- Tyler Akidau, Slava Chernyak, and Reuven Lax, *Streaming Systems: The What, Where, When and How of Large-Scale Data Processing* (Sebastopol, CA: O'Reilly, 2018).
- Martin Kleppmann, *Making Sense of Stream Processing* (Sebastopol, CA: O'Reilly, 2016).
- Martin Kleppmann, *Designing Data-Intensive Applications* (Sebastopol, CA: O'Reilly, 2017).
- Gwen Shapira, Neha Narkhede, and Todd Palino, *Kafka: The Definitive Guide* (Sebastopol, CA: O'Reilly, 2017).
- Michael Nash and Wade Waldron, *Applied Akka Patterns: A Hands-on Guide to Designing Distributed Applications* (Sebastopol, CA: O'Reilly, 2016).
- Jay Kreps, *I Heart Logs* (Sebastopol, CA: O'Reilly, 2014).
- Justin Sheehy, "There Is No Now," *ACM Queue* 13, no. 3 (2015), *https://queue.acm.org/detail.cfm?id=2745385*.

Other O'Reilly-published reports authored by Lightbend engineers and available for free at *lightbend.com/ebooks*:

- Gerard Maas, Stavros Kontopoulos, and Sean Glover, *Designing Fast Data Application Architectures* (Sebastopol, CA: O'Reilly, 2018).
- Boris Lublinsky, *Serving Machine Learning Models: A Guide to Architecture, Stream Processing Engines, and Frameworks* (Sebastopol, CA: O'Reilly, 2017).
- Jonas Bonér, *Reactive Microsystems: The Evolution of Microservices at Scale* (Sebastopol, CA: O'Reilly, 2017).
- Jonas Bonér, *Reactive Microservices Architecture: Design Principles for Distributed Systems* (Sebastopol, CA: O'Reilly, 2016).

- Hugh McKee, *Designing Reactive Systems: The Role of Actors in Distributed Architecture* (Sebastopol, CA: O'Reilly, 2016).

About the Author

Dean Wampler, PhD, is Vice President, Fast Data Engineering, at Lightbend. With over 25 years of experience, Dean has worked across the industry, most recently focused on the exciting big data/fast data ecosystem. Dean is the author of *Programming Scala*, Second Edition, and *Functional Programming for Java Developers*, and the coauthor of *Programming Hive*, all from O'Reilly. Dean is a contributor to several open source projects and a frequent speaker at several industry conferences, some of which he co-organizes, along with several Chicago-based user groups. For more about Dean, visit *deanwampler.com* or find him on Twitter @deanwampler.

Dean would like to thank Stavros Kontopoulos, Luc Bourlier, Debasish Ghosh, Viktor Klang, Jonas Bonér, Markus Eisele, and Marie Beaugureau for helpful feedback on drafts of the two editions of this report.