O'REILLY®

Compliments of Lightbend

Fast Data Architectures for Streaming Applications

Getting Answers Now from Data Sets that Never End

Dean Wampler, PhD

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.

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

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

Editor: Marie Beaugureau
Production Editor: Kristen Brown
Copyeditor: Rachel Head
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2016: First Edition

Revision History for the First Edition

2016-08-31 First Release
2016-10-14 Second Release

978-1-491-97075-1

[LSI]

Table of Contents

1. Introduction. 1
A Brief History of Big Data 2
Batch-Mode Architecture 3

2. The Emergence of Streaming. 5
Streaming Architecture 6
What About the Lambda Architecture? 10

3. Event Logs and Message Queues. 13
The Event Log Is the Core Abstraction 13
Message Queues Are the Core Integration Tool 15
Why Kafka? 17

4. How Do You Analyze Infinite Data Sets?. 19
Which Streaming Engine(s) Should You Use? 23

5. Real-World Systems. 27
Some Specific Recommendations 28

6. Example Application. 31
Machine Learning Considerations 33

7. Where to Go from Here. 37
Additional References 38

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, exclusive reliance on batch-mode processing, where data arrives without immediate extraction of valuable information, is a competitive disadvantage.

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 the streams are finite instead of infinite.

In this report I'll begin with a quick review of the history of big data and batch processing, then discuss how the changing landscape has fueled the emergence of stream-oriented fast data architectures. Next, I'll discuss hallmarks of these architectures and some specific tools available now, focusing on open source options. I'll finish with a look at an example IoT (Internet of Things) application.

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:

1. A scalable and available storage mechanism, such as a distributed filesystem or database
2. A distributed compute engine, for processing and querying the data at scale
3. Tools to manage the resources and services used to implement these systems

Other components layer on top of this core. Big data systems come in two general forms: so-called NoSQL databases that integrate these components into a database system, and more general environments like Hadoop.

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, plus a variety of consistency guarantees. The CAP theorem emerged as a way of understanding the trade-offs between consistency and availability of service 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 evolutionary Cambrian explosion, many of these NoSQL databases have fallen by the wayside, leaving behind a small number of databases 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. 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.

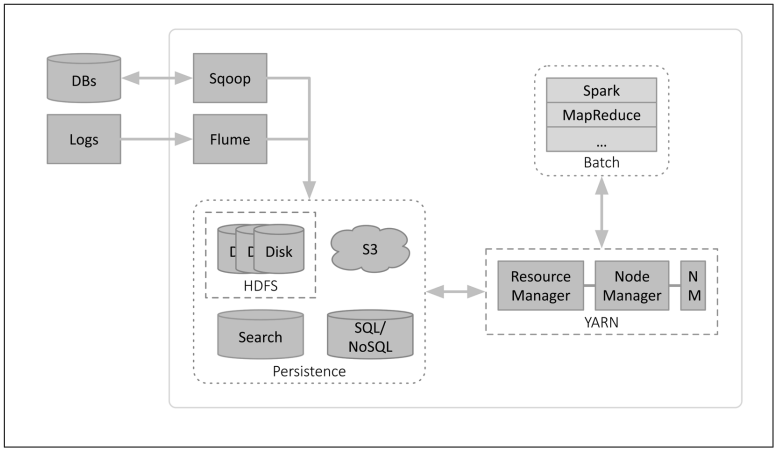

Figure 1-1. Classic Hadoop architecture

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. Functional areas, such as persistence, are indicated by the rounded dotted rectangles.

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, and search engines like Elasticsearch. 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	Many minutes to hours
Job execution times	Minutes to hours

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

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.

Now 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, 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 detecting fraudulent financial activity as it happens, and you can see why streaming is so hot right now.

However, streaming imposes new challenges that go far beyond just making batch systems run faster or more frequently. Streaming introduces new semantics for analytics. It also raises new operational challenges.

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. Of course, I could always add a `WHERE` clause that looks at yesterday's numbers, 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 that network outage that lasted a few hours?

Hence, one of the challenges of streaming is knowing when we can reasonably assume we have all the data for a given context, especially when we want to extract insights as quickly as possible. If data arrives late, we need a way to account for it. Can we get the best of both options, by computing preliminary results now but updating them later if additional data arrives?

Streaming Architecture

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

Figure 2-1 shows such a fast data architecture.

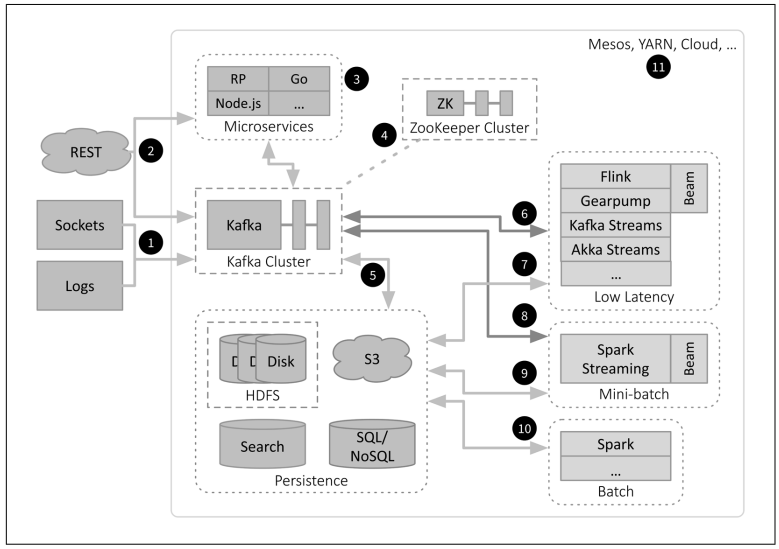

Figure 2-1. Fast data (streaming) architecture

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. I've also suppressed some of the details shown in the previous figure, like the YARN box (see number 11). As before, I still omit specific management and monitoring tools and other possible microservices.

Let's walk through the architecture. Subsequent sections will dive into some of the details:

1. Streams of data arrive into the system over sockets from other servers within the environment or from outside, such as telemetry feeds from IoT devices in the field, social network feeds like the Twitter "firehose," etc. These streams are ingested into a distributed Kafka cluster for scalable, durable, temporary storage. Kafka is the backbone of the architecture. A Kafka cluster will usually have dedicated hardware, which provides maximum load scalability and minimizes the risk of compromised performance due to other 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,[1] as a library on top of Kafka, which also makes it a good first choice for many stream processing chores (see number 6).
2. REST (Representational State Transfer) requests are usually 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. The overhead of REST means it is less common as a high-bandwidth channel for data ingress. Normally it will be used for administration requests, such as for management and monitoring consoles (e.g., Grafana and Kibana). However, REST for data ingress is still supported using custom microservices or through Kafka Connect's REST interface to ingest data into Kafka directly.
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

1 See also Jay Kreps's blog post "Introducing Kafka Streams: Stream Processing Made Simple".

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 and have their own database instances (not shown).

4. Kafka is a distributed system and it uses ZooKeeper (ZK) for tasks requiring consensus, such as leader election, and for storage of some state information. Other components in the environment might also use ZooKeeper for similar purposes. ZooKeeper is deployed as a cluster with its own dedicated hardware, because its demands for system resources, such as disk I/O, would conflict with the demands of other services, such as Kafka's. Using dedicated hardware also protects the ZooKeeper services from being compromised by problems that might occur in other services if they were running on the same machines.
5. Using Kafka Connect, raw data can be persisted directly to longer-term, persistent storage. If some processing is required first, such as filtering and reformatting, then Kafka Streams (see number 6) is an ideal choice. The arrow is two-way because data from long-term storage can be ingested into Kafka to provide a uniform way to feed downstream analytics with data. When choosing between a database or a filesystem, a database is best when row-level access (e.g., CRUD operations) is required. NoSQL provides more flexible storage and query options, consistency vs. availability (CAP) trade-offs, better scalability, and generally lower operating costs, while SQL databases provide richer query semantics, especially for data warehousing scenarios, and stronger consistency. A distributed filesystem or object store, such as HDFS or AWS S3, offers lower cost per GB storage compared to databases and more flexibility for data formats, but they are best used when scans are the dominant access pattern, rather than CRUD operations. Search appliances, like Elasticsearch, are often used to index logs 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 many engines currently vying for attention, most of

which I won't mention here.[2] Flink and Gearpump provide similar rich stream analytics, and both can function as "runners" for dataflows defined with Apache Beam. Akka Streams and Kafka Streams provide the lowest latency and the lowest overhead, but they are oriented less toward building analytics services and more toward building general microservices over streaming data. Hence, they aren't 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., Mesos or YARN; see number 11). No environment would need or want all of these streaming engines. We'll discuss later how to select an appropriate subset. Results from any of these tools can be written back to new Kafka topics or to persistent storage. While it's possible to ingest data directly from input sources into these tools, the durability and reliability of Kafka ingestion, the benefits of a uniform access method, etc. make it the best default choice despite the modest extra overhead. 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, although this imposes longer latency than streaming through Kafka. However, this configuration enables analytics that mix long-term data and stream data, as in the so-called Lambda Architecture (discussed in the next section). Another example is accessing reference data from storage.
8. The mini-batch model of Spark is ideal when longer latencies can be tolerated and the extra window of time is valuable for more expensive calculations, such as training machine learning models using Spark's MLlib or ML libraries or third-party libraries. As before, data can be moved to and from Kafka. Spark Streaming is evolving away from being limited only to mini-batch processing, and will eventually support low-latency streaming too, although this transition will take some time.

2 For a comprehensive list of Apache-based streaming projects, see Ian Hellström's article "An Overview of Apache Streaming Technologies".

Efforts are also underway to implement Spark Streaming support for running Beam dataflows.

9. Similarly, data can be moved between Spark and persistent storage.
10. If you have Spark and a persistent store, like HDFS and/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.
11. All of the above can be deployed to Mesos or Hadoop/YARN clusters, as well as to cloud environments like AWS, Google Cloud Environment, or Microsoft Azure. These environments handle resource management, job scheduling, and more. They offer various trade-offs in terms of flexibility, maturity, additional ecosystem tools, etc., which I won't explore further here.

Let's see where the sweet spots are for streaming jobs as 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)
Time between data arrival and processing	Many minutes to hours	Microseconds to minutes
Job execution times	Minutes to hours	Microseconds to minutes

While the fast data architecture can store the same PB data sets, a streaming job will typically operate on MB to TB at any one time. A TB 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.

What About the Lambda Architecture?

In 2011, Nathan Marz introduced the Lambda Architecture, a hybrid model that uses a batch layer for large-scale analytics over all historical data, a speed layer for low-latency processing of newly arrived data (often with approximate results) and a serving layer to provide a query/view capability that unifies the batch and speed layers.

The fast data architecture we looked at here can support the lambda model, but there are reasons to consider the latter a transitional model.[3] First, without a tool like Spark that can be used to implement 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 the same infrastructure doesn't just unify the batch and speed layers, but batch processing becomes a subset of stream processing. Furthermore, we now know how to achieve the precision we want in streaming calculations, as we'll discuss shortly. Hence, I see the Lambda Architecture as an important transitional step toward fast data architectures like the one discussed here.

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

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

CHAPTER 3

Event Logs and Message Queues

"Everything is a file" is the core, 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 an event log" is the powerful, core abstraction for streaming architectures.

Message queues provide ideal semantics for managing producers writing messages and consumers reading them, thereby joining subsystems together. Implementations can provide durable storage of messages with tunable persistence characteristics and other benefits.

Let's explore these two concepts.

The Event 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 problems they encounter. Log entries usually 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. Well-structured log messages at appropriate execution points are proxies for significant events.

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

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 file before acknowledging the change to clients, after which in-memory data structures and other files with the actual records can be updated with less urgency. 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.

Telemetry from IoT devices
: Almost all widely deployed devices, including cars, phones, network routers, computers, airplane engines, home automation devices, medical devices, kitchen appliances, etc., 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 Apple's Siri for voice recognition. 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 to detect anomalous behavior that may indicate incipient failures, so that proactive action can prevent service disruption.

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

State transitions in a process
: Automated processes, such as manufacturing, chemical processing, etc., 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.

Logs also enable two general architecture patterns: ES (event sourcing), and CQRS (command-query responsibility segregation).

The database WAL is an example of event sourcing. It is a record of all changes (events) that have occurred. The WAL 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 or replicate one to additional copies.

This approach to replication supports CQRS. Having a separate data store for writes ("commands") vs. reads ("queries") enables each one to be tuned and scaled independently, according to its unique characteristics. For example, I might have few high-volume writers, but a large number of occasional readers. Also, 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 the write data stores.[1]

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

Message Queues Are the Core Integration Tool

Message queues are first-in, first-out (FIFO) data structures, which is also the natural way to process logs. Message queues organize data into user-defined topics, where each topic has its own queue. This promotes scalability through parallelism, and it also allows producers (sometimes called writers) and consumers (readers) to focus on the topics of interest. Most implementations allow more than one producer to insert messages and more than one consumer to extract them.

Reading semantics vary with the message queue implementation. In most implementations, when a message is read, it is also deleted from the queue. The queue waits for acknowledgment from the consumer before deleting the message, but this means that policies and enforcement mechanisms are required to handle concurrency cases such as a second consumer polling the queue before the acknowledgment is received. Should the same message be given to the second consumer, effectively implementing *at least once* behavior (see "At Most Once. At Least Once. Exactly Once." on page 16)? Or should the next message in the queue be returned instead, while waiting for the acknowledgment for the first message? What hap-

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".

pens if the acknowledgment for the first message is never received? Presumably a timeout occurs and the first message is made available for a subsequent consumer. But what happens if the messages need to be processed in the same order in which they appear in the queue? In this case the consumers 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 passing information between processes. What should I do if a message fails to arrive? How do I know it failed to arrive in the first place? There are three behaviors we can strive to achieve.

At most once (i.e., "fire and forget") means the message is sent, but the sender doesn't care if it's received or lost. If data loss is not a concern, which might be true for monitoring telemetry, for example, then this model imposes no additional overhead to ensure message delivery, such as requiring acknowledgments from consumers. Hence, it is the easiest and most performant behavior to support.

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 one or more times. This is the most practical model when message loss is not acceptable—e.g., for bank transactions—but duplication can occur.

Exactly once ensures that a message is received once and only once, and is never lost and never repeated. The system must implement whatever mechanisms are required to ensure that a message is received and processed just once. This is the ideal scenario, because it is the easiest to reason about when considering the evolution of system state. It is also impossible to implement in the general case,[2] but it can be successfully implemented for specific cases (at least to a high percentage of reliability[3]).

Often you'll use *at least once* semantics for message transmission, but you'll still want state changes, when present, to be *exactly once* (for example, if you are transmitting transactions for bank

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.

accounts). A deduplication process is required to detect duplicate messages. Most often, a unique identifier of some kind is used: a subsequent message is ignored if it has an identifier that has already been seen. In many contexts, you can exploit *idempotency*, where processing duplicate messages causes no state changes, so they are harmless (other than the processing overhead).

On the other hand, having multiple readers is a way to improve performance through parallelism, but any one reader instance won't see every message in the topic, so the reader must be stateless; it can't know global state about the stream.

Kafka is unique in that messages are not deleted when read, so any number of readers can ingest all the messages in a topic. Instead, Kafka uses either a user-specified retention time (the time to live, or TTL, which defaults to seven days), a maximum number of bytes in the queue (the default is unbounded), or both to know when to delete the oldest messages.

Kafka can't just return the head element for a topic's queue, since it isn't immediately deleted the first time it's read. Instead, Kafka remembers the offset into the topic for each consumer and returns the next message on the next read.

Hence, a Kafka consumer could maintain stream state, since it will see all the messages in the topic. However, since any process might crash, it's necessary to persist any important state changes. One way to do this is to write the current state to another Kafka topic!

Message queues provide many advantages. They decouple producers from consumers, making it easy for them to come and go. They support an arbitrary number of producers and consumers per topic, promoting scalability. They expose a narrow message queue abstraction, which not only makes them easy to use but also effectively hides the implementation so that many scalability and resiliency features can be implemented behind the scenes.

Why Kafka?

Kafka's current popularity is because it is ideally suited as the backbone of fast data architectures. It combines the benefits of event logs as the fundamental abstraction for streaming with the benefits of message queues. The Kafka documentation describes it as "a dis-

tributed, partitioned, replicated commit log service." Note the emphasis on logging, which is why Kafka doesn't delete messages once they've been read. Instead, multiple consumers can see the whole log and process it as they see fit (or even reprocess the log when an analysis task fails). The quote also hints that Kafka topics are partitioned for greater scalability and the partitions can be replicated across the cluster for greater durability and resiliency.

Kafka has also benefited from years of production use and development at LinkedIn, where it started. A year ago, LinkedIn's Kafka infrastructure surpassed 1.1 *trillion* messages a day, and it's still growing.

With the Kafka backbone and persistence options like distributed filesystems and databases, the third key element is the processing engine. For the last several years, Kafka, Cassandra, and Spark Streaming have been a very popular combination for streaming implementations.[4] However, our thinking about stream processing semantics is evolving, too, which has fueled the emergence of Spark competitors.

4 The acronym SMACK has emerged, which adds Mesos and Akka: Spark, Mesos, Akka, Cassandra, and Kafka.

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?

A theory of streaming semantics is emerging to answer questions like these. Central to this theory is the idea that operations like `GROUP BY` and `JOIN` are now based on snapshots of the data available at points in time.

Apache Beam, formerly known as Google Dataflow, is perhaps the best-known mature streaming engine that offers a sophisticated formulation of these semantics. It has become the de facto standard for how precise analytics can be performed in real-world streaming scenarios. A third-party "runner" is required to execute Beam dataflows. In the open source world, teams are implementing this functionality for Flink, Gearpump, and Spark Streaming, while Google's own runner is its cloud service, Cloud Dataflow. This means you will soon be able to write Beam dataflows and run them with these tools, or you will be able to use the native Flink, Gearpump, or Spark Streaming APIs to write dataflows with the same behaviors.

For space reasons, I can only provide a sketch of these semantics here, but two O'Reilly Radar blog posts by Tyler Akidau, a leader of

the Beam/Dataflow team, cover them in depth.[1] If you follow no other links in this report, at least read those blog posts!

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-event processing and what I'll call aggregation processing over many records, including summary statistics and `GROUP BY`, `JOIN`, and similar queries.

Single-event processing is the simplest case to support, where we process each event individually. We might trigger an alarm on an error event, or filter out events that aren't interesting, or transform each event into a more useful format. All of these actions can be performed one event at a time. All of the low-latency tools discussed previously support this simple model.

The next level of sophistication is aggregations of events. Because the stream may be infinite, the simplest approach is to trigger a computation over a fixed-size window (usually limited by time, but possibly also by volume). For example, suppose I want to track clicks per page per minute. At the same time, I might want to segregate all those clicks into different files, one per minute. I collect the data coming in and at the end of each minute, I perform these tasks on the accumulated data, while the next minute's data is being accumulated. This fixed-window approach is what Spark Streaming's mini-batch model provides.[2] It was a clever way to adapt Spark, which started life as a batch-mode tool, to provide a stream processing capability.

But the fixed-window model raises a problem. Time is a first-class concern in streaming. There is a difference between event time (when the event happened on the server where it happened), and processing time (the point in time later when the event was processed, probably on a different server). While processing time is easiest to handle, event time is usually more important for reconstructing reality, such as transaction sequences, anomalous activity, etc. Unfortunately, the fixed-window model I just described works with processing time.

---

1 See "The World Beyond Batch: Streaming 101", and "The World Beyond Batch: Streaming 102".

2 I'm using "windows" and "batches" informally, glossing over the distinction that moving windows over a set of batches is also an important construct.

Complicating the picture is the fact that times across different servers will never be completely aligned. If I'm trying to reconstruct a session—the sequence of events based on event times for activity that spans servers—I have to account for the inevitable clock skew between the servers.

One 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 events 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 Internet, etc.

In the clicks per page per minute example, because of inevitable latencies, however small, the actual events in the mini-batch will include stragglers from the previous window of event time. For example, they might include very late events that were received after a network problem was corrected. Similarly, some events with event times in this window won't arrive until the next mini-batch window, if not much later.

Figure 4-1 illustrates event vs. processing times.

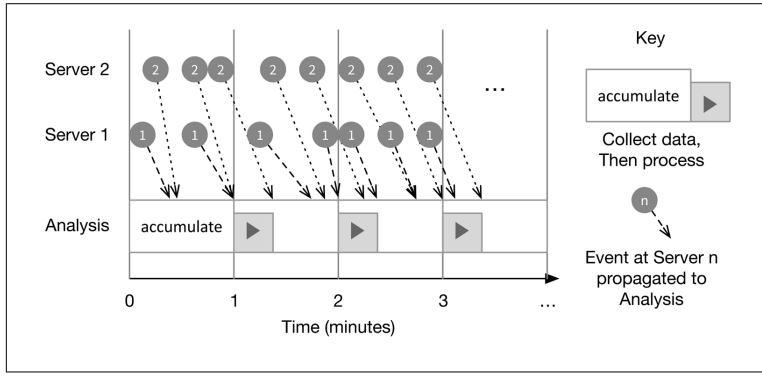

Figure 4-1. Event time vs. processing time

Events generated on Servers 1 and 2 at particular event times propagate to the Analysis server at different rates (shown exaggerated). While most events that occur in a particular one-minute interval arrive in the same interval, some arrive in the next interval. During each minute, events that arrive are accumulated. At minute boundaries, the analysis task for the previous minute's events is invoked, but clearly some data will usually be missing.

So, I really need to process by event time, independent of the mini-batch processing times, to compute clicks per page per minute. However, if my persistent storage supports incremental updates, the second task I mentioned—segregating clicks by minute—is less problematic. When they aren't within the same mini-batch interval, I can just append stragglers later.

This impacts correctness, too. For fast data architectures to truly supplant batch-mode architectures, they have to provide the same correctness guarantees. My clicks per page per minute calculation should have the same accuracy whether I do it incrementally over the event log or in a batch job over yesterday's accumulated data.

I still need some sense of finite windows, because I can't wait for an infinite stream to end. Therefore, I need to decide when to trigger processing. Three concepts help us refine what to do within windows:

- *Watermarks* indicate that all events relative to a session or other "context" have been received, so it's now safe to process the final result for the context.
- *Triggers* are a more general tool for invoking processing, including the detection of watermarks, a timer firing, a threshold being reached, etc. They may result in computation of incomplete results for a given context, as more data may arrive later.
- *Accumulation modes* encapsulate how we should 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?

Except for the case when we know we've seen everything for a context, we still need a plan for handling late-arriving events. One possibility is to structure our analysis so that we can always take a snapshot in time, then improve it as events arrive later. My dashboard might show results for clicks per page per minute for the last hour, but some of them might be updated as late events arrive and are processed. In a dashboard, it's probably better to show provisional results sooner rather than later, as long as it's clear to the viewer 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 their bookkeeping; instead, they add a correction entry later to fix earlier mistakes. Similarly, my streaming tools need to support retractions, followed by new values.

We've just scratched the surface of the important considerations required for processing infinite streams in a timely fashion, yet 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 options and how they support these concepts.

Which Streaming Engine(s) Should You Use?

How do the low-latency and mini-batch engines available today stand up? Which ones should you use?

While Spark Streaming version 1.x was limited to mini-batch intervals based on processing time, *Structured Streaming* in Spark version 2.0 starts the evolution toward a low-latency streaming engine with full support for the semantics just discussed. Spark 2.0 still uses mini-batches internally, but features like event-time support have been added. 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. The project also wants to simplify common scenarios where other services integrate with streaming data.

Today, Spark Streaming offers the benefit of enabling very rich, very scalable, and resource-intensive processing, like training machine learning models, when the longer latencies of mini-batches are tolerable.

Flink and Gearpump are the two most prominent open source tools that are aggressively pursuing the ability to run Beam dataflows. Flink is better known, while Gearpump is better integrated with Akka and Akka Streams. For example, Gearpump can *materialize* Akka Streams (analogous to being a Beam runner) in a distributed configuration not natively supported by Akka. Cloudera is leading an effort to add Beam support to Spark's Structured Streaming as well.

Kafka Streams and Akka Streams are designed for the sweet spot of building microservices that process data asynchronously, versus a

focus on data analytics. Both provide single-event processing with very low latency and high throughput, as well as some aggregation processing, like grouping and joins. Akka and Akka Streams are also very good for implementing complex event processing (CEP) scenarios, such as lifecycle state transitions, alarm triggering off certain events, etc. Kafka Streams runs as a library on top of the Kafka cluster, so it requires no additional cluster setup and minimal additional management overhead. Akka is part of the Lightbend Reactive Platform, which puts more emphasis on building microservices. Therefore, Akka Streams is a good choice when you are integrating a wide variety of other services with your data processing system.

The reason why I included so many tools in Figure 2-1 is because a real environment may need several of them. However, there's enough overlap that you won't need all of them. Indeed, you shouldn't take on the management burden of using all of these tools. If you already use or plan to 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.

If you need the sophisticated semantics and low latency provided by Beam, then use the Beam API with Flink or Gearpump as the runner, or use the Flink or Gearpump native API to define your dataflows directly. Even if you don't (yet) need Beam semantics, if you are primarily doing stream processing and you don't need the full spectrum of Spark features, consider using Flink or Gearpump instead of Spark.

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.

Use Akka Streams to implement and integrate more general microservices and a wide variety of other tools and products, with flexible options for communication protocols besides REST, such as Aeron, and especially those that support the *Reactive Streams* specification,

a new standard for asynchronous communications.[3] Consider Gearpump as a materializer for Akka Streams for more deployment options. Akka Streams also supports the construction of complex graphs of streams for sophisticated dataflows. Other recommended uses include low-latency processing (similar to Kafka Streams) such as ETL filtering and transformation, alarm signaling, and two-way stateful sessions, such as interactions with IoT devices.

3 Reactive Streams is a protocol for dynamic flow control though *backpressure* that's negotiated between the consumer and producer.

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, network partitions, etc. Leveraging replication prevents data loss and enables a service to keep going using the remaining instances. Leveraging isolation prevents cascading failures.

Elastic
: You can expect the load to vary considerably over the lifetime of a service. It's essential to implement dynamic, automatic scalability, both up and down, based on load.

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.

Batch-mode and interactive systems have traditionally had less stringent requirements for these qualities. Fast data architectures are just like other online systems where downtime and data loss are serious, costly problems. When implementing these architectures, developers who have focused on analytics tools that run in the back office are suddenly forced to learn new skills for distributed systems programming and operations.

Some Specific Recommendations

Most of the components we've discussed strive to 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, etc., which I won't repeat here. However, here are some specific recommendations:

- Ingest all inbound data into Kafka first, then consume it with the stream processors. For all the reasons I've highlighted, you get durable, scalable, resilient storage as well as multiple consumers, replay capabilities, etc. You also get the uniform simplicity and power of event log and message queue semantics as the core concepts 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.
- Because Kafka Streams leverages the distributed management features of Kafka, you should use it when you can to add processing capabilities with minimal additional management overhead in your architecture.
- For integration microservices, use Reactive Streams–compliant protocols for direct message and data exchange, for the resilient capabilities of backpressure as a flow-control mechanism.
- Use Mesos, YARN, or a similarly mature management infrastructure for processes and resources, 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 wisely (if you need them). 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.
- Seek professional production support for your environment, even when using open source solutions. It's cheap insurance and it saves you time (which is money).

CHAPTER 6
Example Application

Let's finish with a look at an example application, IoT telemetry ingestion and anomaly detection for home automation systems. Figure 6-1 labels how the parts of the fast data architecture are used. I've grayed out less-important details from Figure 2-1, but left them in place for continuity.

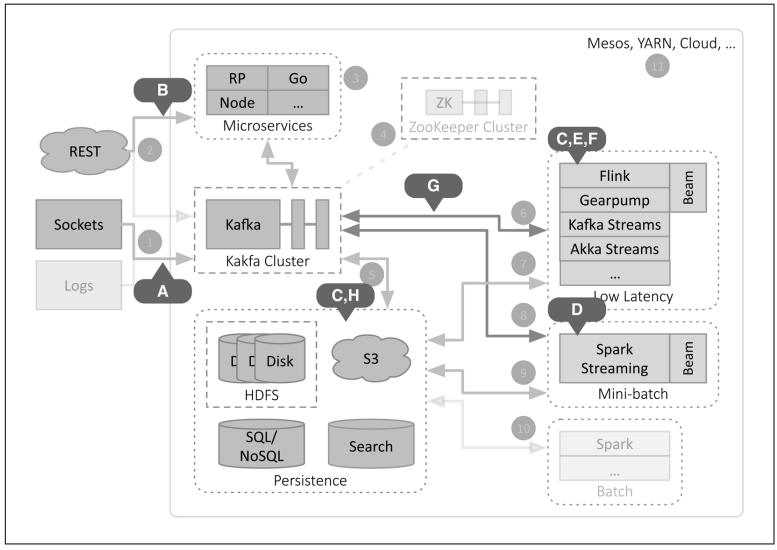

Figure 6-1. IoT example

Let's look at the details of this figure:

A. Stream telemetry data from devices into Kafka. This includes low-level machine and operating system metrics, such as temperatures of components like hard drive controllers and CPUs, CPU and memory utilization, etc. Application-specific metrics are also included, such as service requests, user interactions, state transitions, and so on. Various logs from remote devices and microservices will also be ingested, but they are grayed out here to highlight what's unique about this example.

B. Mediate two-way sessions between devices in the field and microservices over REST. These sessions process user requests to adjust room temperatures, control lighting and door locks, program timers and time-of-day transitions (like raising the room temperature in the morning), and invoke services (like processing voice commands). Sessions can also perform administration functions like downloading software patches and even shutting down a device if problems are detected. Using one Akka Actor per device is an ideal way to mirror a device's state within the microservice and use the network of Akka Actors to mirror the real topology. Because Akka Actors are so lightweight (you can run millions of them concurrently on a single laptop, for example), they scale very well for a large device network.

C. Clean, filter, and transform telemetry data and session data to convenient formats using Kafka Streams, and write it to storage using Kafka Connect. Most data is written to HDFS or S3, where it can be subsequently scanned for analysis, such as for machine learning, aggregations over time windows, dashboards, etc. The data may be written to databases and/or search engines if more focused queries are needed.

D. Train machine learning models "online", as data arrives, using Spark Streaming (see the next section for more details). Spark Streaming can also be used for very large-scale streaming aggregations where longer latencies are tolerable.

E. Apply the latest machine learning models to "score" the data in real time and trigger appropriate responses using Flink, Gearpump, Akka Streams, or Kafka Streams. Use the same streaming engine to serve operational dashboards via Kafka (see G).

Compute other running analytics. If sophisticated Beam-style stream processing is required, choose Flink or Gearpump.

F. Use Akka Streams for complex event processing, such as alarming.

G. Write stream processing results to special topics in Kafka for low-latency, downstream consumption. Examples include the following:

   a. Updates to machine learning model parameters

   b. Alerts for anomalous behavior, which will be consumed by microservices for triggering corrective actions on the device, notifying administrators or customers, etc.

   c. Data feeds for dashboards

   d. State changes in the stream for durable retention, in case it's necessary to restart a processor and recover the last state of the stream

   e. Buffering of analytics results for subsequent storage in the persistence tier (see H)

H. Store results of stream processing.

Other fast data deployments will have broadly similar characteristics.

Machine Learning Considerations

Machine learning has emerged as a product or service differentiator. Here are some applications of it for our IoT scenario:

Anomaly detection
: Look for outliers and other indications of anomalies in the telemetry data. For example, hardware telemetry can be analyzed to predict potential hardware failures so that services can be performed proactively and to detect atypical activity that might indicate hardware tampering, software hacking, or even a burglary in progress.

Voice interface
: Respond to voice commands for service.

Image classification
: Alert the customer when people or animals are detected in the environment using images from system cameras.

Recommendations
: Recommend service features to customers that reflect their usage patterns and interests.

Automatic tuning of the IoT environment
: In a very large network of devices and services, usage patterns can change dramatically during the day, and during certain times of year. Usage spikes are common. Hence, being able to automatically tune how services are distributed and allocated to devices, how and when devices interact with services, etc. makes the overall system more robust.

There are nontrivial aspects of deploying machine learning. Like stream processing in general, incremental training of machine learning models has become important. For example, if you are detecting spam, ideally you want your model to reflect the latest kinds of spam, not a snapshot from some earlier time. The term "online" is used for machine learning algorithms where training is incremental, often per-datum. Online algorithms were invented for training with very large data sets, where "all at once" training is impractical. However, they have proven useful for streaming applications, too.

Many algorithms are compute-intensive enough that they take too long for low-latency situations. In this case, Spark Streaming's mini-batch model is ideal for striking a balance, trading off longer latencies for the ability to use more sophisticated algorithms.

When you're training a model with Spark Streaming mini-batches, you can apply the model to the mini-batch data at the same time. This is the simplest approach, but sometimes you need to separate training from scoring. For robustness reasons, you might prefer "separation of concerns," where a Spark Streaming job focuses only on training and other jobs handle scoring. Then, if the Spark Streaming job crashes, you can continue to score data with a separate process. You also might require low-latency scoring, for which Spark Streaming is currently ill suited.

This raises the problem of how models can be shared between these processes, which may be implemented in very different tools. There are a few ways to share models:

1. Implement the underlying model (e.g., logistic regression) in both places, but share parameters. Duplicate implementations won't be an easy option for sophisticated machine learning models, unless the implementation is in a library that can be shared. If this isn't an issue, then the parameters can be shared in one of two ways:
   a. The trainer streams updates to individual model parameters to a Kafka topic. The scorer consumes the stream and applies the updates.
   b. The trainer writes changed parameters to a database, perhaps all at once, from which the scorer periodically reads them.
2. Use a third-party machine learning service, either hosted or on-premise,[1] to provide both training and scoring in one place that is accessible from different streaming jobs.

Note that moving model parameters between jobs means there will be a small delay where the scoring engine has slightly obsolete parameters. Usually this isn't a significant concern.

1 E.g., Deeplearning4J.

CHAPTER 7
Where to Go from Here

Fast data is the natural evolution of big data to be stream oriented and quickly processed, 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. Form your own opinions and let me know what you discover.[1] At Lightbend, we've been working hard to build tools, techniques, and expertise to help our customers succeed with fast data. Please take a look.

1 I'm at *dean.wampler@lightbend.com* and on Twitter, @deanwampler.

Additional References

Besides the links throughout this report, the following references are very good for further exploration:

1. Justin Sheehy, "There Is No Now," *ACM Queue*, Vol. 13, Issue 3, March 10, 2015, *https://queue.acm.org/detail.cfm?id=2745385*.
2. Jay Kreps, *I Heart Logs*, September 2014, O'Reilly.
3. Martin Kleppmann, *Making Sense of Stream Processing*, May 2016, O'Reilly.
4. Martin Kleppmann, *Designing Data-Intensive Applications*, currently in Early Release, O'Reilly.

About the Author

Dean Wampler, PhD (@deanwampler), is Lightbend's architect for fast data products in the office of the CTO. With over 25 years of experience, he's 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 he is a frequent speaker at several industry conferences, some of which he co-organizes, along with several Chicago-based user groups.

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 this report.