Chapter 2: Introducing Apache Kafka

Apache Kafka (or simply Kafka) is an event streaming platform. But it is also more than that. It is an entire ecosystem of technologies designed to assist in the construction of complete event-driven systems. Kafka goes above and beyond the essential set of event streaming capabilities, providing rich event persistence, transformation, and processing semantics.

Event streaming platforms are a comparatively recent paradigm within the broader message-oriented middleware class. There are only a handful of mainstream implementations available, compared to hundreds of MQ-style brokers, some going back to the 1980s (for example, Tuxedo). Compared to established messaging standards such as AMQP, MQTT, XMPP, and JMS, there are no equivalent standards in the streaming space. Kafka is a leader in the area of event streaming, and more broadly, event-driven architecture. While there is no *de jure* standard in event streaming, Kafka is the benchmark to which most competing products orient themselves. To this effect, several competitors — such as Azure Event Hubs and Apache Pulsar — offer APIs that mimic Kafka.

Event streaming platforms are an active area of continuous research and experimentation. In spite of this, event streaming platforms aren't just a niche concept or an academic idea with few esoteric use cases; they can be applied effectively to a broad range of messaging and eventing scenarios, routinely displacing their more traditional counterparts.

Kafka is written in Java, meaning it can run comfortably on most operating systems and hardware configurations. It can equally be deployed on bare metal, in the Cloud, and a Kubernetes cluster. And finally, Kafka has libraries written for just about every programming language, meaning that virtually every developer can start taking advantage of event streaming and push their application architecture to the next level of resilience and scalability.

The history of Kafka

Apache Kafka was originally developed by LinkedIn, and was subsequently open-sourced in early 2011. The name 'Kafka' was chosen by one of its founders — Jay Kreps. Kreps chose to name the software after the famous 20th-century author Franz Kafka because it was "a system optimised for writing". Kafka gained the full Apache Software Foundation project status in October 2012, having graduated from the Apache Incubator program.

Kafka was born out of a need to track and process large volumes of site events, such as page views and user actions, as well as for the aggregation log data. Before Kafka, LinkedIn maintained several disparate data pipelines, which presented a challenge from both complexity and operational scalability perspectives. In July 2011, having consolidated the individual platforms, Kafka was

processing approximately one billion events per day. By 2012, this number had risen to 20 billion. By July 2013, Kafka was carrying 200 billion events per day. Two years later, in 2015, Kafka was turning over one trillion events per day, with peaks of up to 4.5 million events per second.

Over the four years of 2011 to 2015, the volume of records has grown by three orders of magnitude. By the end of this period, LinkedIn was moving well over a petabyte of event data per week. By all means, this level of growth could not be attributed to Kafka alone; however, Kafka was undoubtedly a key enabler from an infrastructure perspective.

As of October 2019, LinkedIn maintains over 100 Kafka clusters, comprising more than 4,000 brokers. These collectively serve more than 100,000 topics and 7 million partitions. The total number of records handled by Kafka has surpassed 7 trillion per day.

The present day

The industry adoption of Kafka has been nothing short of phenomenal. The list of tech giants that heavily rely on Kafka is impressive in itself. To name just a few:

- **Yahoo** uses Kafka for real-time analytics, handling up to 20 gigabits of uncompressed event data per second in 2015. Yahoo is also a major contributor to the Kafka ecosystem, having open-sourced its in-house Cluster Manager for Apache Kafka (CMAK) product.
- **Twitter** heavily relies on Kafka for its mobile application performance management and analytics product, which has been clocked at five billion sessions per day in February 2015. Twitter processes this stream using a combination of Apache Storm, Hadoop, and AWS Elastic MapReduce.
- **Netflix** uses Kafka as the messaging backbone for its Keystone pipeline — a unified event publishing, collection, and routing infrastructure for both batch and stream processing. As of 2016, Keystone comprises over 4,000 brokers deployed entirely in the Cloud, which collectively handle more than 700 billion events per day.
- **Tumblr** relies on Kafka as an integral part of its event processing pipeline, capturing up 500 million page views a day back in 2012.
- **Square** uses Kafka as the underlying bus to facilitate stream processing, website activity tracking, metrics collection and monitoring, log aggregation, real-time analytics, and complex event processing.
- **Pinterest** employs Kafka for its real-time advertising platform, with 100 clusters comprising over 2,000 brokers deployed in AWS. Pinterest is turning over in excess of 800 billion events per day, peaking at 15 million per second.
- **Uber** is among the most prominent of Kafka adopters, processing in excess of a trillion events per day — mostly for data ingestion, event stream processing, database changelogs, log aggregation, and general-purpose publish-subscribe message exchanges. In addition, Uber is an avid open-source contributor — having released its in-house cluster replication solution *uReplicator* into the wild.

And it's not just the engineering-focused organisations that have adopted Kafka — by some estimates, up a third of Fortune 500 companies use Kafka to fulfill their event streaming and processing needs.

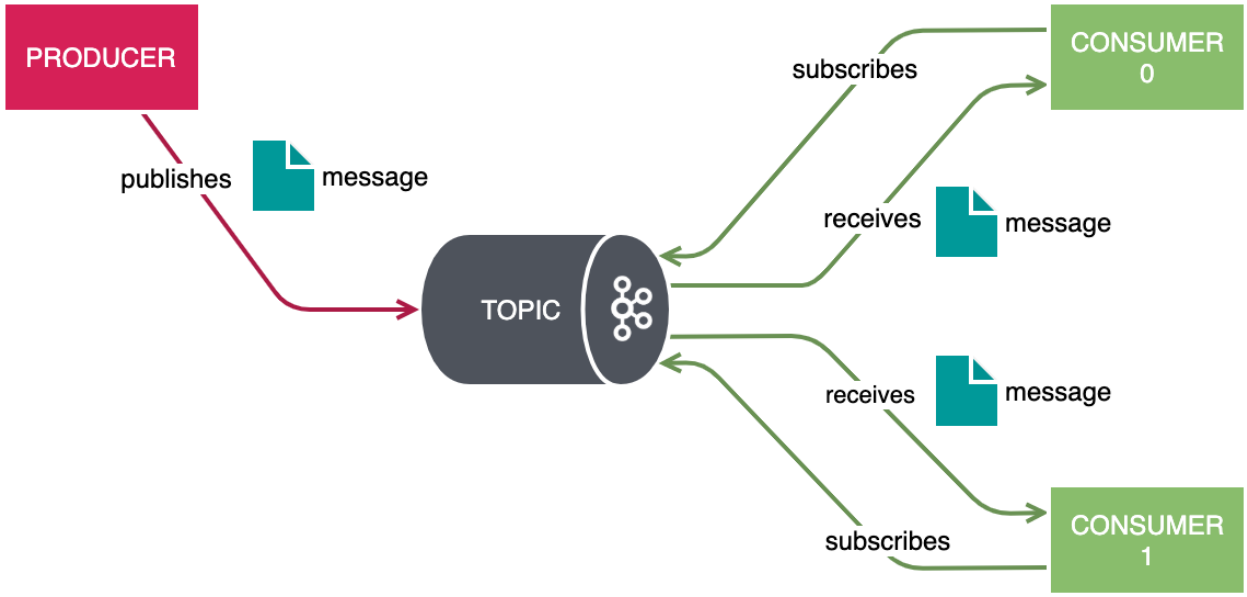
There are good reasons for this level of industry adoption. As it happens, Kafka is one of the most well-supported and well-regarded event streaming platforms, boasting an impressive number of open-source projects that integrate with Kafka. Some of the big names include Apache Storm, Apache Flink, Apache Hadoop, LogStash and the Elasticsearch Stack, to name a few There are also Kafka Connect integrations with every major SQL database, and most NoSQL ones too. At the time of writing, there are circa one hundred supported off-the-shelf connectors, which does not include custom connectors that have been independently developed.

Uses of Kafka

Chapter 1: Event Streaming Fundamentals has provided the necessary background, fitting Kafka as an event streaming platform within a larger event-driven system.

There are several use cases falling within the scope of EDA that are well-served by Apache Kafka. This section covers some of these scenarios, illustrating how Kafka may be used to address them.

Publish-subscribe



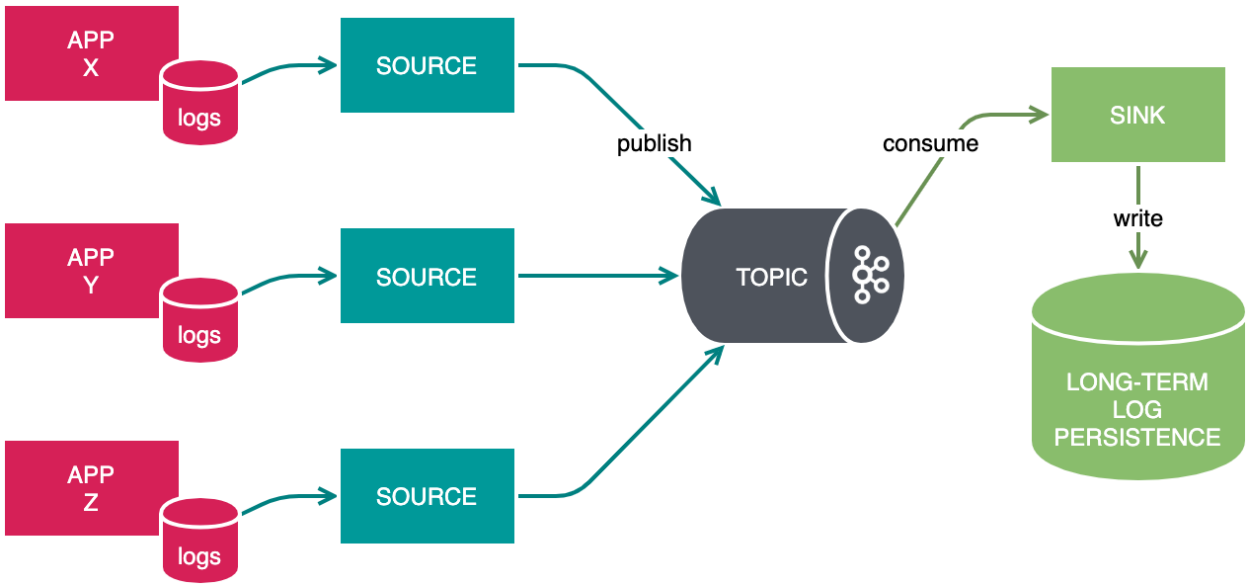
Pub-Sub

Any messaging scenario where producers are generally unaware of consumers, and instead publish messages to well-known aggregations called *topics*. Conversely, consumers are generally unaware of the producers but are instead concerned with specific content categories. The producer and consumer

ecosystems are loosely-coupled, being aware of only the common topic(s) and messaging schema(s). This pattern is commonly used in the construction of loosely-coupled microservices.

When Kafka is used for general-purpose publish-subscribe messaging, it will be competing with its 'enterprise' counterparts, such as message brokers and service buses. Admittedly, Kafka might not have all the features of some of these middleware platforms — such as message deletion, priority levels, producer flow control, distributed transactions, or dead-letter queues. On the other hand, these features are mostly representative of traditional messaging paradigms — intrinsic to how these platforms are commonly used. Kafka works in its own idiomatic way — optimised around unbounded sequences of immutable events. As long as a publish-subscribe relationship can be represented as such, then Kafka is fit for the task.

Log aggregation

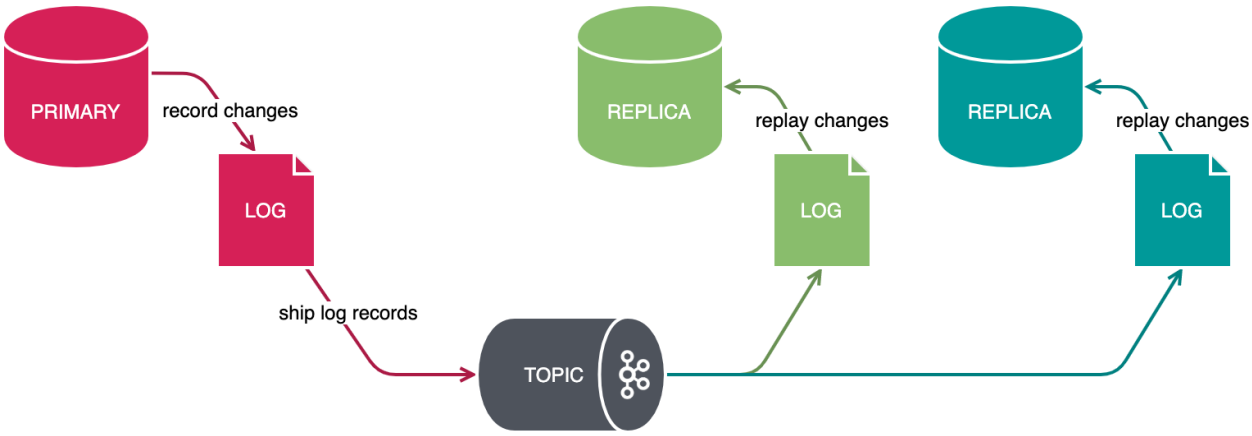


Log aggregation

Dealing with large volumes of log-structured events, typically emitted by application or infrastructure components. Logs may be generated at burst rates that significantly outstrip the ability of query-centric datastores to keep up with log ingestion and indexing, which are regarded as 'expensive' operations. Kafka can act as a buffer, offering an intermediate, durable datastore. The ingestion process will act as a sink, eventually collating the logs into a read-optimised database (for example, Elasticsearch or HBase).

A log aggregation pipeline may also contain intermediate steps, each adding value en route to the final destination; for example, to compress log data, encrypt log content, normalise the logs into a canonical form, or sanitise the log entries — scrubbing them of personally-identifiable information.

Log shipping



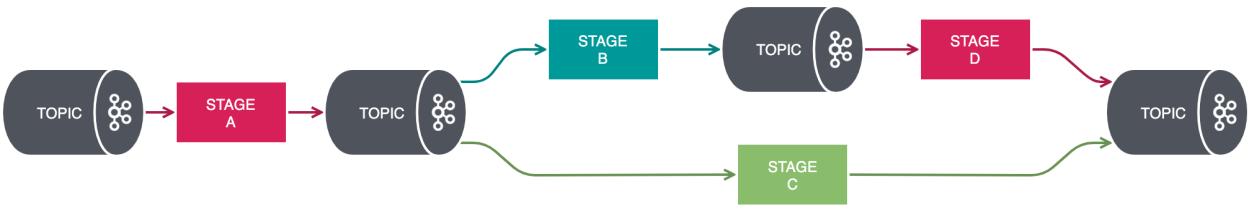
Log shipping

While sounding vaguely similar to log aggregation, the shipping of logs is a vastly different concept. Essentially, this involves the real-time copying of journal entries from a master data-centric system to one or more read-only replicas. Assuming stage changes are fully captured as journal records, replaying those records allows the replicas to accurately mimic the state of the master, albeit with some lag.

Kafka's optional ability to partition records within a topic to create independent, causally ordered sequences of events allows for replicas to operate in one of *sequential* or *causal* consistency models — depending on the chosen partitioning scheme. The various consistency models were briefly covered in Chapter 1: Event Streaming Fundamentals. Both consistency models are sufficient for creating read-only copies of the original data.

Log shipping is a key enabler for another related architectural pattern — *event sourcing*. Kafka will act as a durable event store, allowing any number of consumers to rebuild a point-in-time snapshot of their application state by replaying all records up to that point in time. Loss of state information in any of the downstream consumers can be recovered by replaying the events from the last stable checkpoint, thereby reducing the need to take frequent backups.

SEDA pipelines



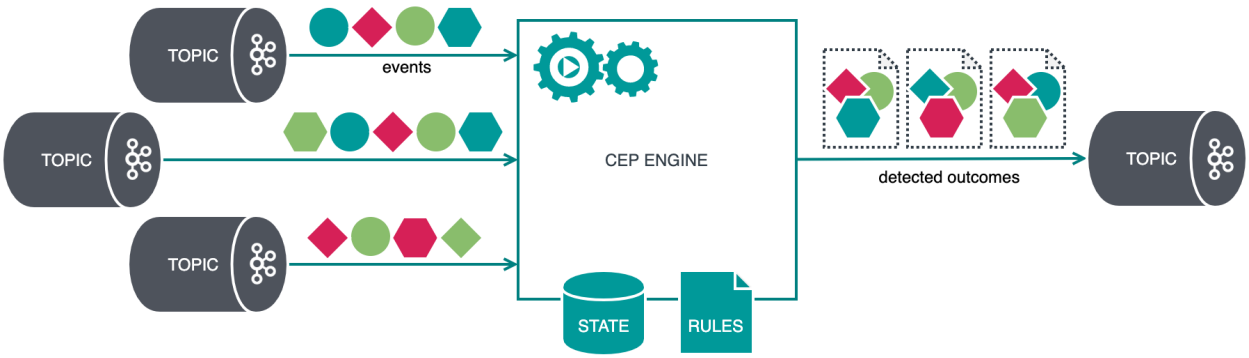
SEDA pipeline

Staged Event-Driven Architecture (SEDA) is the application of pipelining to event-oriented data. Events flow unidirectionally through a series of processing stages linked by topics, each one performing a mapping operation before publishing a transformed event to the next topic. Intermediate stages simultaneously act as both consumers and producers, and may scale autonomously and independently of one another to match their unique load demands. By breaking a complex problem into stages, SEDA improves the modularity of the system.

A SEDA pipeline may combine fan-in and fan-out topologies. Stages may consume events from multiple topics simultaneously, performing the equivalent of an SQL `JOIN` on event streams. Stages can also publish to multiple topics, feeding several downstream pipelines.

As a pattern, SEDA is readily found in data warehousing, data lakes, reporting, analytics, and other Business Intelligence systems, and is often a crucial element of Big Data applications. SEDA can also be used in log aggregation; in fact, log aggregation is a narrow specialisation of SEDA.

CEP

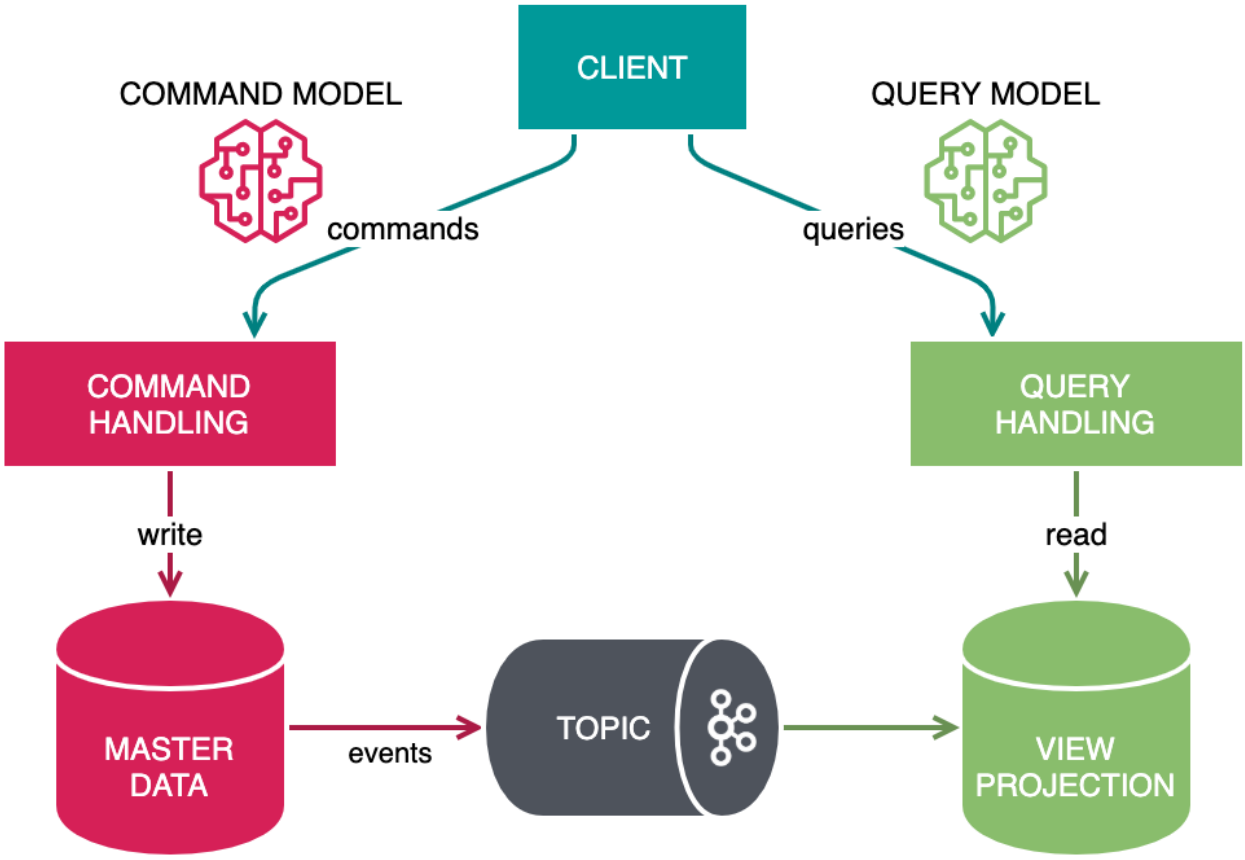


CEP

Complex Event Processing (CEP) extracts meaningful information and patterns in a stream of discrete events, or across a set of disjoint event streams. CEP processors tend to be stateful, as they must be able to efficiently recall prior events to identify patterns that might span a broad timeframe, ranging from milliseconds to days, depending on the context.

CEP is heavily employed in such applications as algorithmic stock trading, security threat analysis, real-time fraud detection, and control systems.

Event-sourced CQRS



CQRS

Command-Query Responsibility Segregation (CQRS) separates the actions that mutate state from the actions that query the state. Because mutations and queries typically exhibit contrasting runtime characteristics and require vastly different, often contradictory optimisation decisions, the separation of these concerns is conducive to building highly performant systems. The flip side is complexity — requiring multiple datastores and duplication of data — each datastore will maintain a unique projection of the master dataset, built from a dedicated data pipe. Kafka curbs some of the complexity inherent in CQRS architectures by acting as a common event-sourced ledger, using the concept of *consumer groups* to individually feed the different query-centric datastores.

This pattern is related to log shipping, and might appear identical at first glance. The differences are subtle. Log shipping is performed on low-level, internal representations of data, where both the replicas and the master datastore are coupled to, and share the same internal data structures. Put differently, log shipping is an internal mechanism that shuttles data within the confines of a single domain. In comparison, CQRS assumes disparate systems and spans domain boundaries. All parties are coupled to some canonical representation of events, which is versioned independently of the parties' internal representations of those events.

This chapter has introduced the reader to Apache Kafka — the world's most recognised and widely deployed event streaming platform. We looked at the history behind Kafka — how it started its journey and what it has become.

We also explored the various use cases that Kafka comfortably enables and supports. These are vast and varied, demonstrating Kafka's overall flexibility and eagerness to cater to a diverse range of event streaming scenarios.