Thorough Introduction to Apache Kafka™

hackernoon.com/thorough-introduction-to-apache-kafka-6fbf2989bbc1

A deep dive into a system that serves as the heart of many companies' architecture

Some of the monoliths who use Apache Kafka

Introduction

Kafka is a word that gets heard a lot nowadays... A lot of leading digital companies seem to use it as well. But what is it actually?

Kafka was originally developed at LinkedIn in 2011 and has improved a lot since then. Nowadays it is a whole platform, allowing you to redundantly store absurd amounts of data, have a message bus with huge throughput (millions/sec) and use real-time stream processing on the data that goes through it all at once.

This is all well and great, but stripped down to its core, Kafka is a distributed, horizontally-scalable, fault-tolerant, commit log.

Those were some fancy words, let's go at them one by one and see what they mean. Afterwards, we will dive deep into how it works.

Distributed

A distributed system is one which is split into multiple running machines, all of which work together in a cluster to appear as one single node to the end user. Kafka is distributed in the sense that it stores, receives and sends messages on different nodes (called brokers).

> I have a Thorough Introduction on this as well

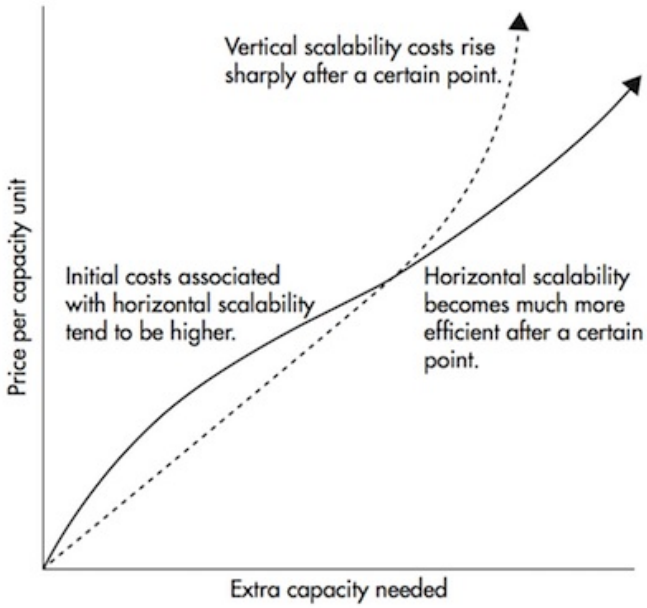
The benefits to this approach are high scalability and fault-tolerance.

Horizontally-scalable

Let's define the term vertical scalability first. Say, for instance, you have a traditional database server which is starting to get overloaded. The way to get this solved is to simply increase the resources (CPU, RAM, SSD) on the server. This is called **vertical scaling**—where you add more resources to the machine. There are two big disadvantages to scaling upwards:

1. There are limits defined by the hardware. You cannot scale upwards indefinitely.
2. It usually requires downtime, something which big corporations cannot afford.

Horizontal scalability is solving the same problem by throwing more machines at it. Adding a new machine does not require downtime nor are there any limits to the amount of machines you can have in your cluster. The catch is that not all systems support horizontal scalability, as they are not designed to work in a cluster and those that are are usually more complex to work with.



Horizontal scaling becomes **much cheaper** after a certain threshold

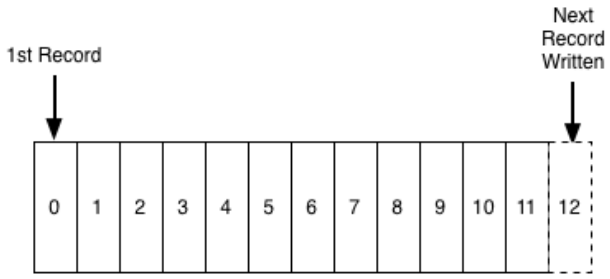
Fault-tolerant

Something that emerges in non-distributed systems is that they have a single point of failure (SPoF). If your single database server fails (as machines do) for whatever reason, you're screwed.

Distributed systems are designed in such a way to accommodate failures in a configurable way. In a 5-node Kafka cluster, you can have it continue working even if 2 of the nodes are down. It is worth noting that fault-tolerance is at a direct tradeoff with performance, as in the more fault-tolerant your system is, the less performant it is.

Commit Log

A commit log (also referred to as write-ahead log, transaction log) is a persistent ordered data structure which only supports appends. You cannot modify nor delete records from it. It is read from left to right and guarantees item ordering.



Sample illustration of a commit log, taken from here

- Are you telling me that Kafka is such a simple data structure?

In many ways, yes. This structure is at the heart of Kafka and is invaluable, as it provides ordering, which in turn provides deterministic processing. Both of which are non-trivial problems in distributed systems.

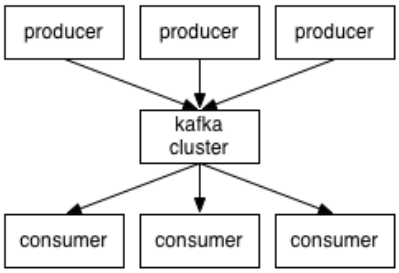
Kafka actually stores all of its messages to disk (more on that later) and having them ordered in the structure lets it take advantage of sequential disk reads.

- Reads and writes are a constant time O(1) *(knowing the record ID)*, which compared to other structure's O(log N) operations on disk is a huge advantage, as each disk seek is expensive.
- Reads and writes do not affect another. Writing would not lock reading and vice-versa (as opposed to balanced trees)

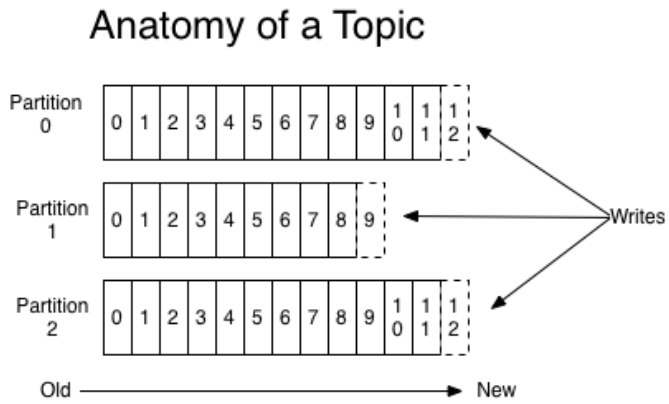
These two points have huge performance benefits, since the data size is completely decoupled from performance. Kafka has the same performance whether you have 100KB or 100TB of data on your server.

How does it work?

Applications (**producers**) send messages (**records**) to a Kafka node (**broker**) and said messages are processed by other applications called **consumers**. Said messages get stored in a **topic** and consumers subscribe to the topic to receive new messages.

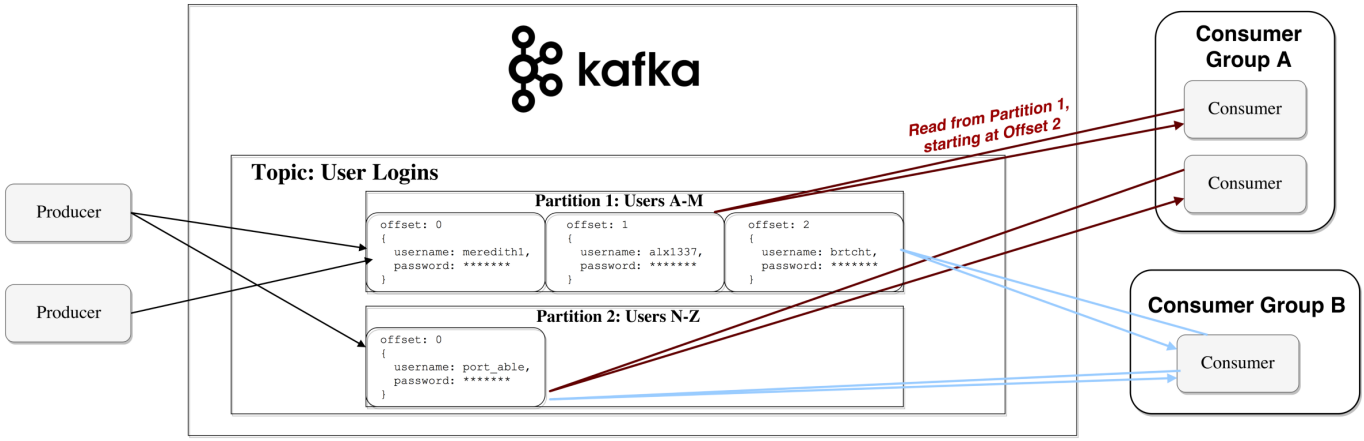


As topics can get quite big, they get split into **partitions** of a smaller size for better performance and scalability. (ex: *say you were storing user login requests, you could split them by the first character of the user's username)*
Kafka guarantees that all messages inside a partition are ordered in the sequence they came in. The way you distinct a specific message is through its **offset**, which you could look at as a normal array index, a sequence number which is incremented for each new message in a partition.



Kafka follows the principle of a dumb broker and smart consumer. This means that Kafka does not keep track of what records are read by the consumer and delete them but rather stores them a set amount of time (e.g one day) or until some size threshold is met. Consumers themselves poll Kafka for new messages and say what records they want to read. This allows them to increment/decrement the offset they're at as they wish, thus being able to replay and reprocess events.

It is worth noting that consumers are actually consumer groups which have one or more consumer processes inside. In order to avoid two processes reading the same message twice, each partition is tied to only one consumer process per group.



Representation of the data flow

Persistence to Disk

As I mentioned earlier, Kafka actually stores all of its records to disk and does not keep anything in RAM. You might be wondering how this is in the slightest way a sane choice. There are numerous optimizations behind this that make it feasible:

1. Kafka has a protocol which groups messages together. This allows network requests to group messages together and reduce network overhead, the server in turn persist chunk of messages in one go and consumer fetch large linear chunks at once
2. Linear reads/writes on a disk are fast. The concept that modern disks are slow is because of numerous disk seeks, something that is not an issue in big linear operations.
3. Said linear operations are heavily optimized by the OS, via **read-ahead** (prefetch large block multiples) and **write-behind** (group small logical writes into big physical writes) techniques.
4. Modern OSes cache the disk in free RAM. This is called **pagecache**.
5. Since Kafka stores messages in a standardized binary format unmodified throughout the whole flow *(producer->broker->consumer)*, it can make use of the **zero-copy** optimization. That is when the OS copies data from the pagecache directly to a socket, effectively bypassing the Kafka broker application entirely.

All of these optimizations allow Kafka to deliver messages at near network speed.

Data Distribution & Replication

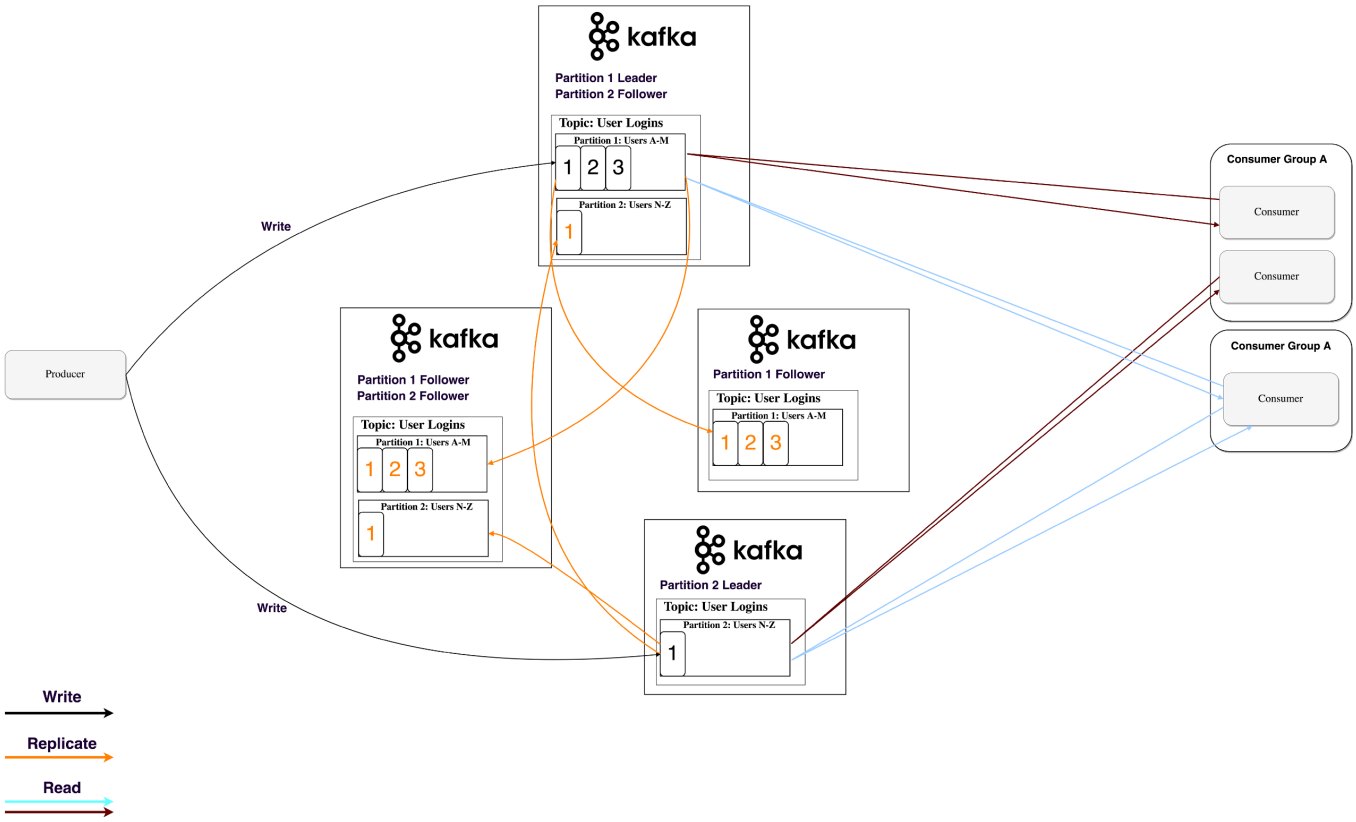
Let's talk about how Kafka achieves fault-tolerance and how it distributes data between nodes.

Data Replication

Partition data is replicated across multiple brokers in order to preserve the data in case one broker dies.

At all times, one broker *"owns"* a partition and is the node through which applications write/read from the partition. This is called a **partition leader**. It replicates the data it receives to ***N*** other brokers, called **followers**. They store the data as well and are ready to be elected as leader in case the leader node dies.

This helps you configure the guarantee that any successfully published message will not be lost. Having the option to change the replication factor lets you trade performance for stronger durability guarantees, depending on the criticality of the data.



4 Kafka brokers with a replication factor of 3

In this way, if one leader ever fails, a follower can take his place.

You may be asking, though:

- How does a producer/consumer know who the leader of a partition is?

For a producer/consumer to write/read from a partition, they need to know its leader, right? This information needs to be available from somewhere.
Kafka stores such metadata in a service called **Zookeeper**.

What is Zookeeper?

Zookeeper is a distributed key-value store. It is highly-optimized for reads but writes are slower. It is most commonly used to store metadata and handle the mechanics of clustering (heartbeats, distributing updates/configurations, etc).

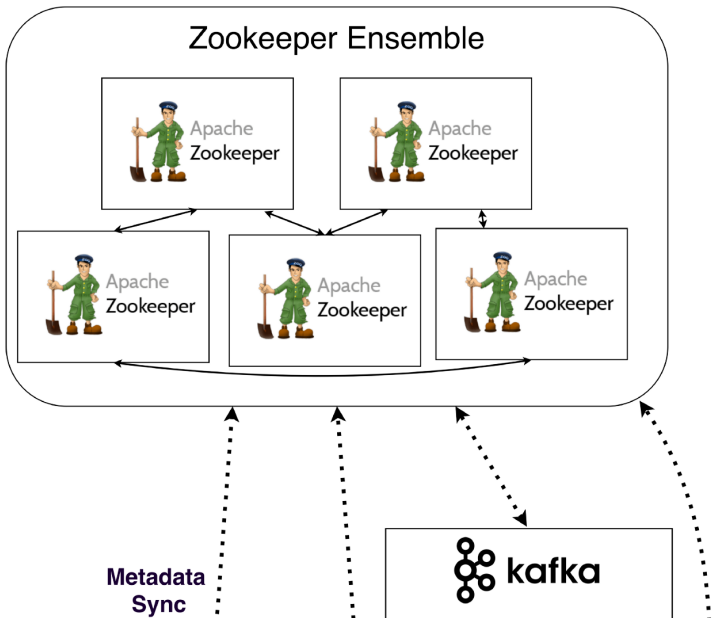
It allows clients of the service (the Kafka brokers) to subscribe and have changes sent to them once they happen. This is how brokers know when to switch partition leaders. Zookeeper is also extremely fault-tolerant and it ought to be, as Kafka heavily depends on it.

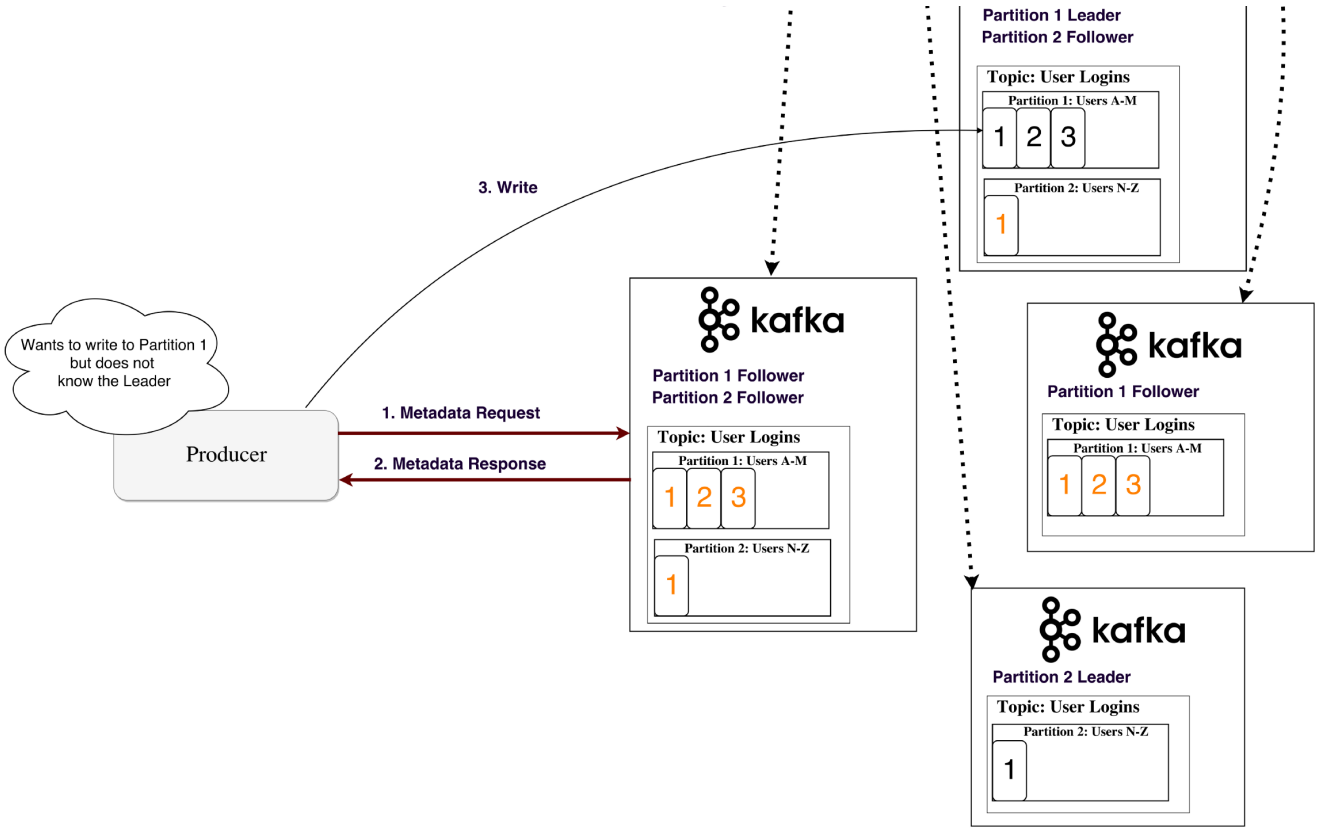
It is used for storing all sort of metadata, to mention some:

- Consumer group's offset per partition (although modern clients store offsets in a separate Kafka topic)
- ACL (Access Control Lists)—used for limiting access/authorization
- Producer & Consumer Quotas —maximum message/sec boundaries
- Partition Leaders and their health

How does a producer/consumer know who the leader of a partition is?

Producer and Consumers used to directly connect and talk to Zookeeper to get this (and other) information. Kafka has been moving away from this coupling and since versions 0.8 and 0.9 respectively, clients fetch metadata information from Kafka brokers directly, who themselves talk to Zookeeper.





Metadata Flow

Streaming

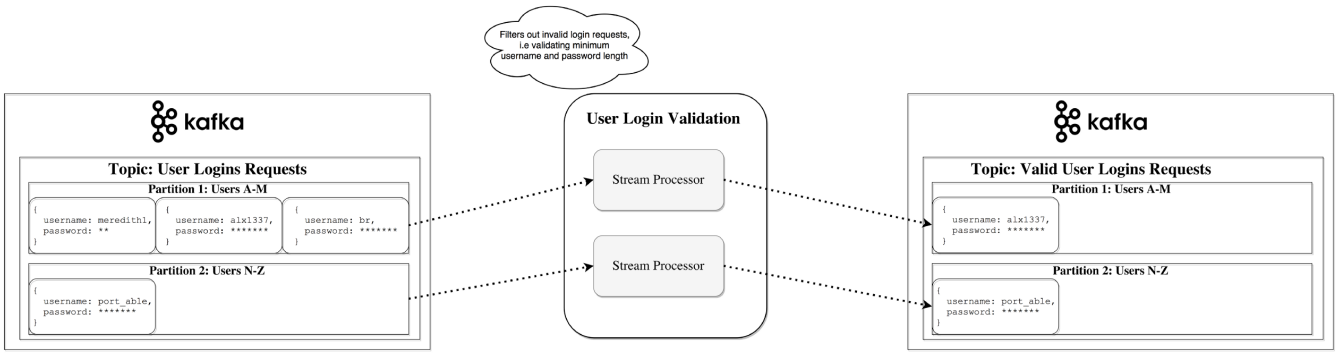
In Kafka, a stream processor is anything that takes continual streams of data from input topics, performs some processing on this input and produces a stream of data to output topics (or external services, databases, the trash bin, wherever really...)

It is possible to do simple processing directly with the producer/consumer APIs, however for more complex transformations like joining streams together, Kafka provides a integrated Streams API library.

This API is intended to be used within your own codebase, it is not running on a broker. It works similar to the consumer API and helps you scale out the stream processing work over multiple applications (similar to consumer groups).

Stateless Processing

A stateless processing of a stream is deterministic processing that does not depend on anything external. You know that for any given data you will always produce the same output independent of anything else. An example for that would be simple data transformation — appending something to a string `"Hello"` -> `"Hello, World!"`.



Stream-Table Duality

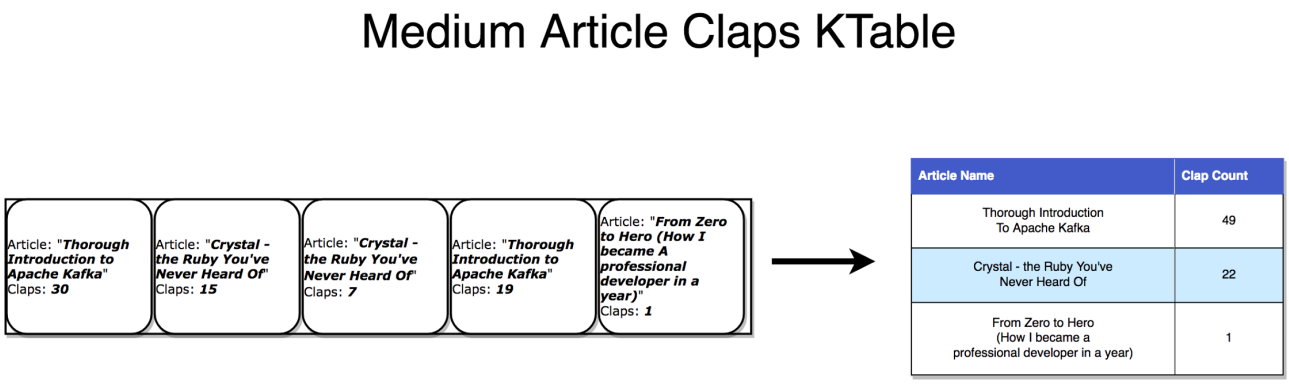
It is important to recognize that streams and tables are essentially the same. A stream can be interpreted as a table and a table can be interpreted as a stream.

Stream as a Table

A stream can be interpreted as a series of updates for data, in which the aggregate is the final result of the table. This technique is called ***Event Sourcing*.**

If you look at how synchronous database replication is achieved, you'll see that it is through the so-called **streaming replication**, where each change in a table is sent to a replica server. Another example of event sourcing is Blockchain ledgers—a ledger is a series of changes as well.

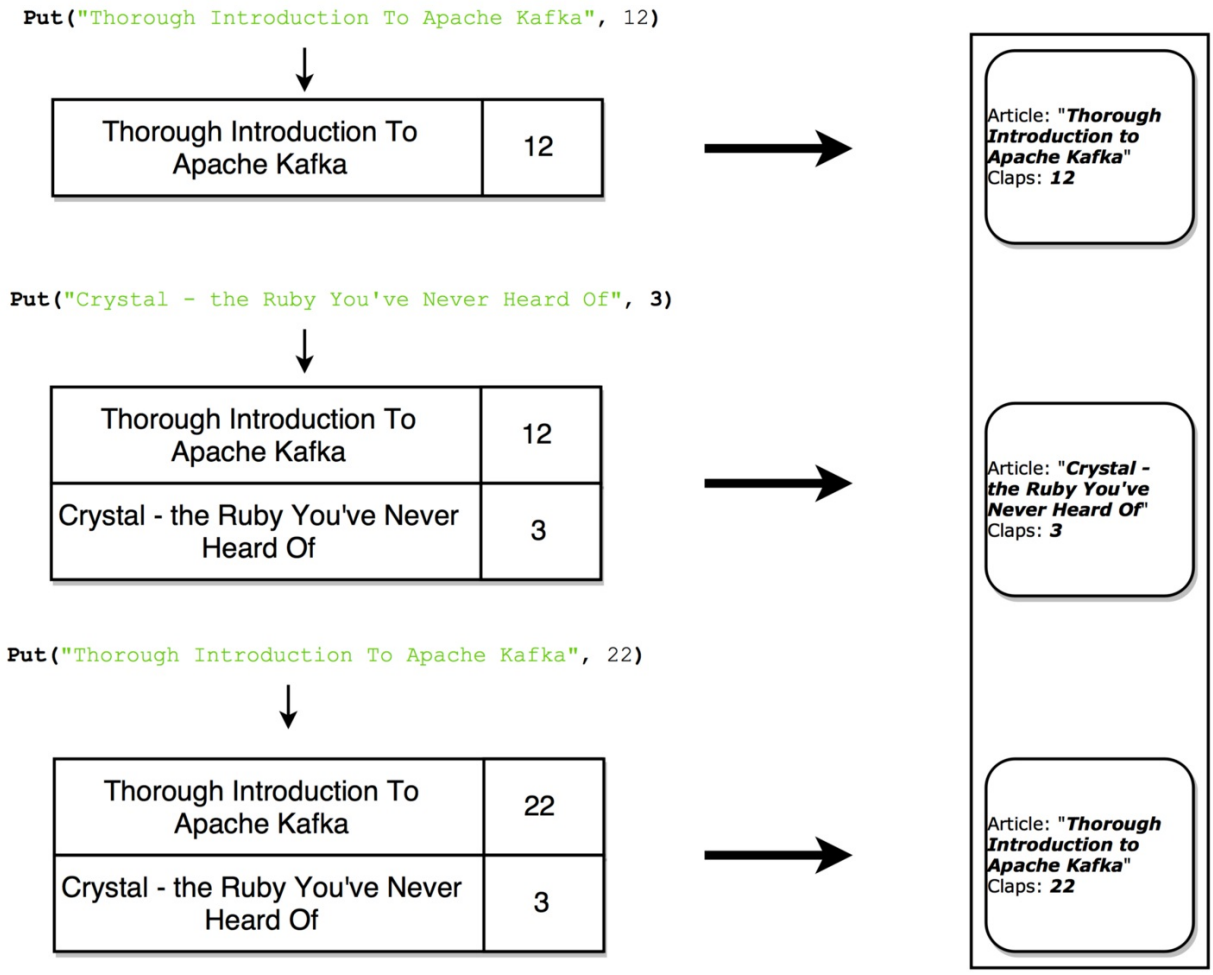
A Kafka stream can be interpreted in the same way —events which when accumulated form the final state. Such stream aggregations get saved in a local RocksDB (by default) and are called a **KTable.**



Each record increments the aggregated count

Table as a Stream

A table can be looked at as a snapshot of the latest value for each key in a stream. In the same way stream records can produce a table, table updates can produce a changelog stream.



Each update produces a snapshot record in the stream

Stateful Processing

Some simple operations like `map()` or `filter()` are stateless and do not require you to keep any data regarding the processing. However, in real life, most operations you'll do will be stateful (e.g `count()`) and as such will require you to store the currently accumulated state.

The problem with maintaining state on stream processors is that the stream processors can fail! Where would you need to keep this state in order to be fault-tolerant?

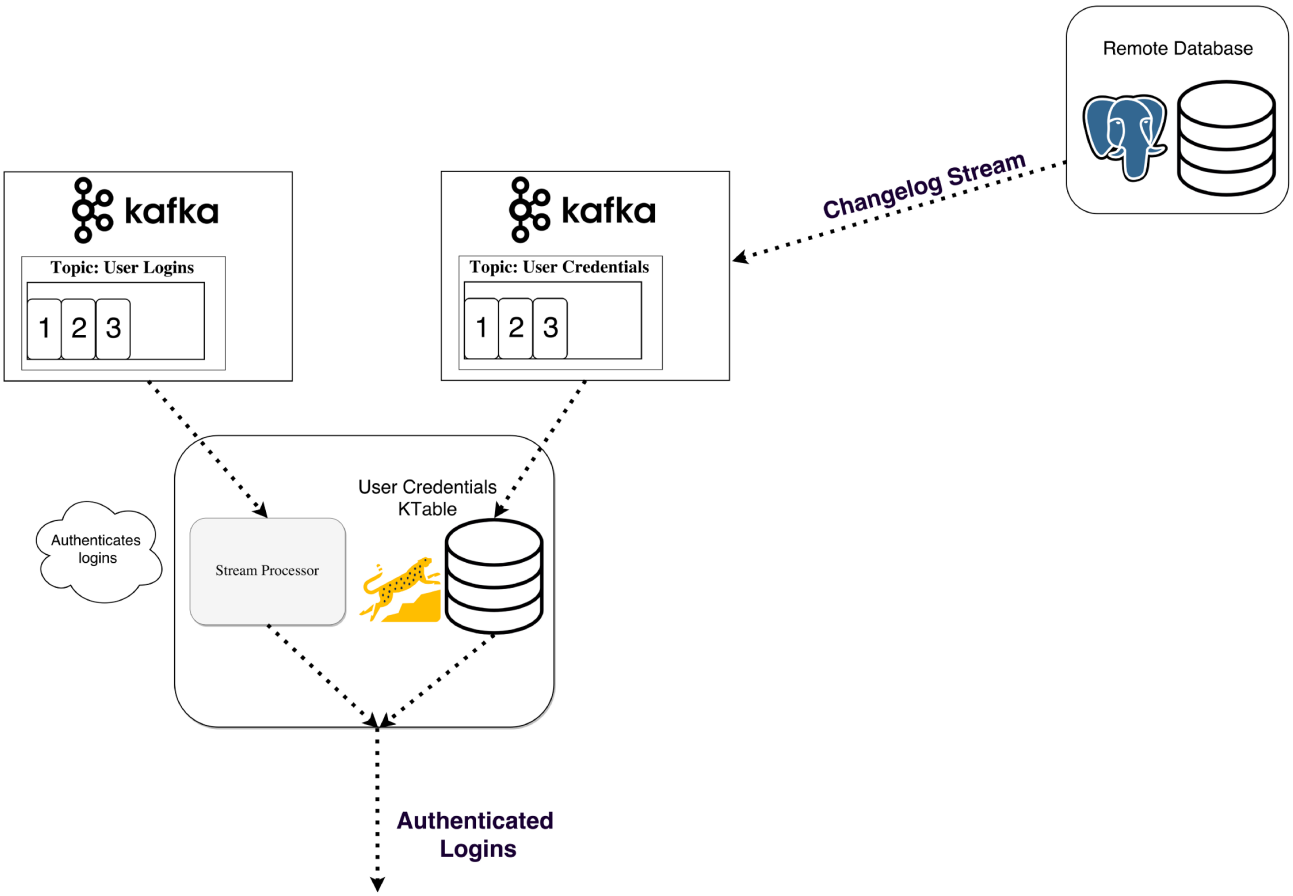
A naive approach is to simply store all state in a remote database and join over the network to that store. The problem with this is that there is no locality of data and lots of network round-trips, both of which will significantly slow down your application. A more subtle but important problem is that your stream processing job's uptime would be tightly coupled to the remote database and the job will not be self-contained *(a change in the database from another team might break your processing)*.

So what's a better approach?
Recall the duality of tables and streams. This allows us to convert streams into tables that are co-located with our processing. It also provides us with a mechanism for handling fault

tolerance — by storing the streams in a Kafka broker.

A stream processor can keep its state in a local table (e.g RocksDB), which will be updated from an input stream (after perhaps some arbitrary transformation). When the process fails, it can restore its data by replaying the stream.

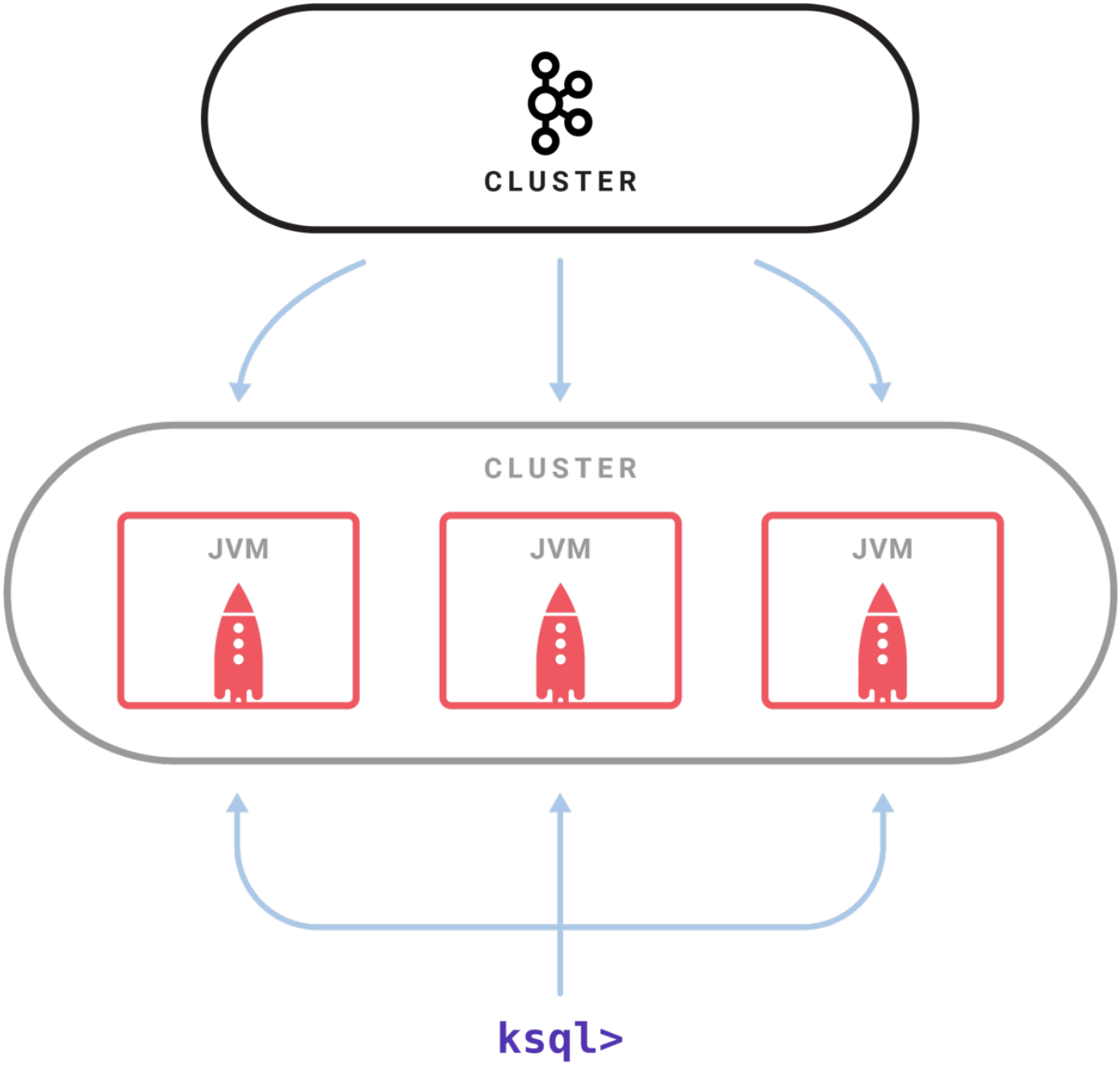
You could even have a remote database be the producer of the stream, effectively broadcasting a changelog with which you rebuild the table locally.



Stateful processing, joining a KStream with a KTable

KSQL

Normally, you'd be forced to write your stream processing in a JVM language, as that is where the only official Kafka Streams API client is.



Sample KSQL setup

You set up a KSQL server and interactively query it through a CLI to manage the processing. It works with the same abstractions (KStream & KTable), guarantees the same benefits of the Streams API (scalability, fault-tolerance) and greatly simplifies work with streams.

This might not sound as a lot but in practice is way more useful for testing out stuff and even allows people outside of development (e.g product owners) to play around with stream processing. I encourage you to take a look at the quick-start video and see how simple it is .

Streaming alternatives

Kafka streams is a perfect mix of power and simplicity. It arguably has the best capabilities for stream jobs on the market and it integrates with Kafka way easier than other stream processing alternatives (**Storm**, **Samza**, **Spark**, **Wallaroo**).

The problem with most other stream processing frameworks is that they are complex to work with and deploy. A batch processing framework like Spark needs to:

- Control a large number of jobs over a pool of machines and efficiently distribute them across the cluster.
- To achieve this it has to dynamically package up your code and physically deploy it to the nodes that will execute it. (along with configuration, libraries, etc.)

Unfortunately tackling these problems makes the frameworks pretty invasive. They want to control many aspects of how code is deployed, configured, monitored, and packaged.

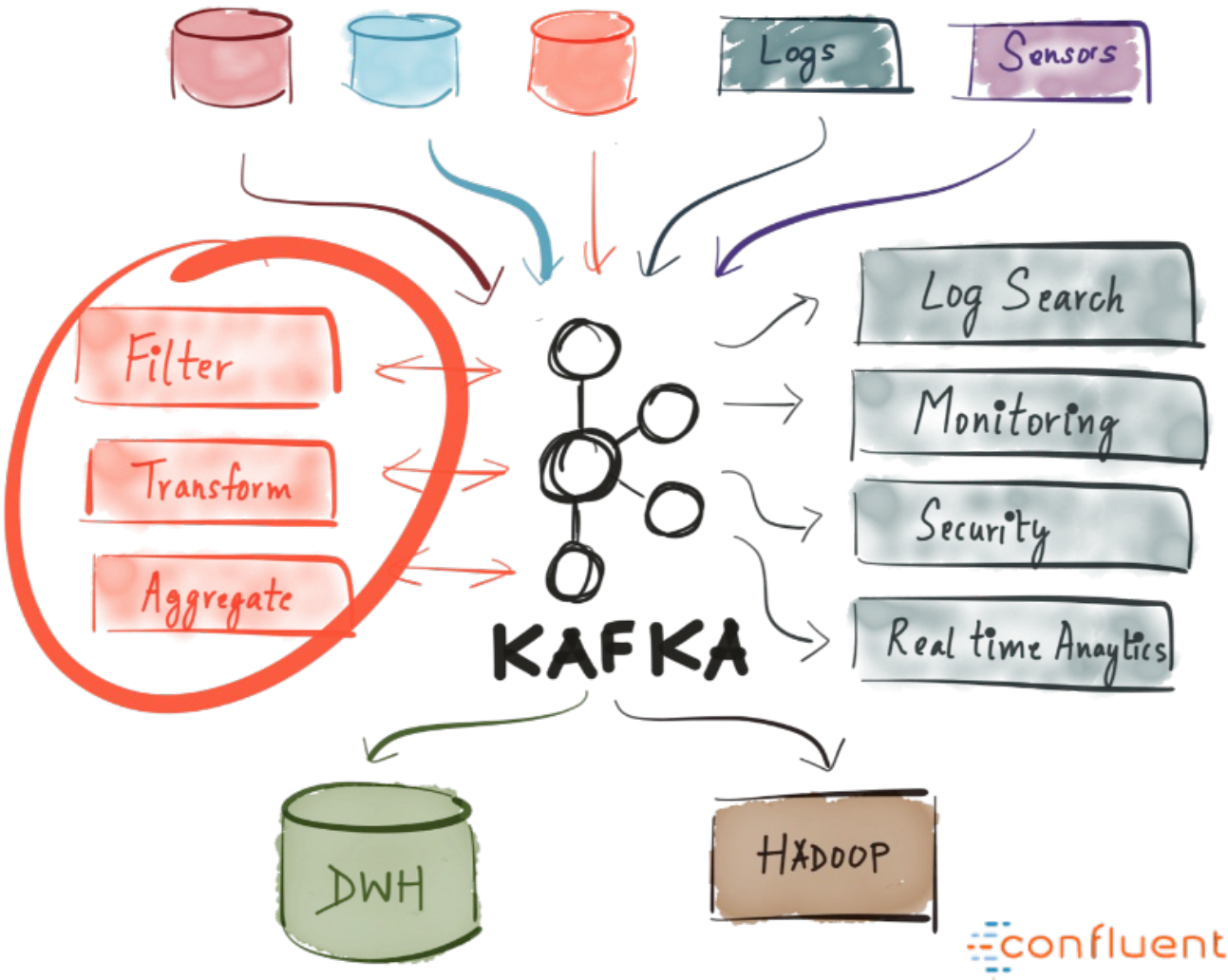
Kafka Streams let you roll out your own deployment strategy when you need it, be it Kubernetes, Mesos, Nomad, Docker Swarm or others.

The underlying motivation of Kafka Streams is to enable all your applications to do stream processing without the operational complexity of running and maintaining yet another cluster. The only potential downside is that it is tightly coupled with Kafka, but in the modern world where most if not all real-time processing is powered by Kafka that may not be a big disadvantage.

When would you use Kafka?

As we already covered, Kafka allows you to have a huge amount of messages go through a centralized medium and store them without worrying about things like performance or data loss.

This means it is perfect for use as the heart of your system's architecture, acting as a centralized medium that connects different applications. Kafka can be the center piece of an event-driven architecture and allows you to truly decouple applications from one another.



Kafka allows you to easily decouple communication between different (micro)services. With the Streams API, it is now easier than ever to write business logic which enriches Kafka topic data for service consumption. The possibilities are huge and I urge you to explore how companies are using Kafka.

Why has it seen so much use?

High performance, availability and scalability alone are not strong enough reasons for a company to adopt a new technology. There are other systems which boast similar properties, but none have become so widely used. Why is that?

The reason Kafka has grown in popularity (and continues to do so) is one key thing—businesses nowadays benefit greatly from event-driven architecture. This is because the world has changed—an enormous (and ever-growing) amount of data is being produced and consumed by many different services (Internet of Things, Machine Learning, Mobile, Microservices).

A single real-time event broadcasting platform with durable storage is the cleanest way to achieve such an architecture. Imagine what kind of a mess it would be if streaming data to/from each service used a different technology specifically catered to it.

This, paired with the fact that Kafka provides the appropriate characteristics for such a generalized system *(durable storage, event broadcast, table and stream primitives, abstraction via KSQL, open-source, actively developed)* make it an obvious choice for companies.

Summary

Apache Kafka is a distributed streaming platform capable of handling trillions of events a day. Kafka provides low-latency, high-throughput, fault-tolerant publish and subscribe pipelines and is able to process streams of events.

We went over its basic semantics (producer, broker, consumer, topic), learned about some of its optimizations (pagecache), learned how it's fault-tolerant by replicating data and were introduced to its ever-growing powerful streaming abilities.

Kafka has seen large adoption at thousands of companies worldwide, including a third of the Fortune 500. With the active development of Kafka and the recently released first major version 1.0 *(1st November, 2017),* there are predictions that this Streaming Platform is going to be as big and central of a data platform as relational databases are.

I hope that this introduction helped familiarize you with Apache Kafka and its potential.

Further Reading Resources & Things I did not mention

The rabbit hole goes deeper than this article was able to cover. Here are some features I did not get the chance to mention but are nevertheless important to know:

Controller Broker, in-sync replicas — The way in which Kafka keeps the cluster healthy and ensures adequate consistency and durability.

Connector API — API helping you connect various services to Kafka as a source or sink (PostgreSQL, Redis, ElasticSearch)

Log Compaction — An optimization which reduces log size. Extremely useful in changelog streams

Exactly-once Message Semantics — Guarantee that messages are received exactly once. This is a big deal, as it is difficult to achieve.

Resources

Apache Kafka's Distributed Systems Firefighter — the Controller Broker — Another blog post of mine where I dive into how coordination between the broker works and much more.

Confluent Blog — a wealth of information regarding Apache Kafka

Kafka Documentation — Great, extensive, high-quality documentation

Kafka Summit 2017 videos

Thank you for taking the time to read this.

If you think this information was helpful, please consider giving it a hefty amount of claps to increase its visibility and help new people find it!

~Stanislav Kozlovski

Update

This article opened the door for me to join Confluent. I am immensely grateful for the opportunity they have given me — I currently work on Kafka itself, which is beyond awesome! Confluent is a big data company founded by the creators of Apache Kafka themselves! We currently work on the whole Kafka ecosystem, including a managed Kafka-as-a-service cloud offering.

We are hiring for a lot of positions (especially SRE/Software Engineers) in Europe and the USA! If you are interested in working on Kafka itself, looking for new opportunities or just plain curious — make sure to message me on and I will share all the great perks that come from working in a bay area company.