***Sharding Technique:***

we cannot build a scalable model without understanding the **Sharding Technique.**

The Sharding is technique of dividing the large DB into small DB. Each Small DB is called Shards.

There is multiple technique of dividing the DB in small shards.

1. **Horizontal Sharding**:

It is range based partitioning. Example: Table Location. If we have divided DB into 3 Shards. If Pin Code is less then 5K, we will store Location in 1st Shards, for Pin Code between 5k-10k in second Shards and grater than 10k in Third Shards.

This kind of Sharding is called Horizontal Sharing.

Problem: If the field/column based on which horizontal Sharding is done, in exam pin code, And If that field is not chosen properly.

For Example : If Pin Code is always grater than 10k, then all the load/request will be coming to 3rd partition only

1. **Vertical Sharding:**

In this Sharding, DB is divided based on the feature.

For exam: User Module or feature: 1 DB, For AAIS 1 DB, For IP\_FootPrint another DB etc.

Problem : If any one feature is widely used as compared to other features ,in that case DB with widely used feature can have more request coming in as compare to other DB.

So If application experiences the additional growth, than it may be necessary to further divide or partition feature specific DB.

1. **Directory based Sharding: Lookup Server**

**It** is widely used Sharding technique. It is very efficient in rebalancing.

Here one directory server is kept in front of all partitioned DB. This Directory server is server which known current partitioning scheme.

So Directory server know where particular data resides. So First we connect to directory server, then directory server will lookup data/table in any of the existing partitioned DB.

1. **Hashing based Sharding:**

This is very rarely used Sharding, where partitioning is done based on hash function.

Problem: If we need to add few more DB partition, then we need to change Hash function also.

**Sharding cons :**

1. **Joins and Denormalization :** One DB is partitioned and spread across multiple server, It is not feasible to perform Join operation.
2. **Referential Integrity:** If one table is present in one partition in any server, and foreign key table resides in another machine or db in different server. Then referencing becomes very complex in case of Sharding.

**SQL/ NOSQL**

SQL :

It has schema, fixed table structure, Normalization concept like every piece of data stored in DB only once, and other places are connected/uses any other data via relationship etc.

Limitation:

1. Number of columns in a table is fixed. As it has fixed schema, then it is not possible that for one table, one row has different number of columns and other row has different number column.
2. We need to have a clear schema to relation DB table. And some time because of these relations, SQL becomes too complex and slow.

**No-SQL : No fixed Schema**

1. Used in Scalable product as it can store a huge amount of data.

Type:

1. **Key-value DB** : Redis uses this kind of the DB

How does a key-value database work?

A key-value database, aka *key-value store*, associates a value (which can be anything from a number or simple string, to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a Database Management System (DBMS).

Key-value databases use compact, efficient index structures to be able to quickly and reliably locate a value by its key, making them ideal for systems that need to be able to find and retrieve data in constant time. Redis, for instance, is a key-value database that is optimized for tracking relatively simple data structures (primitive types, lists, heaps, and maps) in a persistent database. By only supporting a limited number of value types, Redis is able to expose an extremely simple interface to querying and manipulating them, and when configured optimally is capable of extremely high throughput.

1. **Document DB:**

A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents. Document databases make it easier for developers to store and query data in a database by using the same document-model format they use in their application code. The flexible, semistructured, and hierarchical nature of documents and document databases allows them to evolve with applications’ needs. The document model works well with use cases such as catalogs, user profiles, and content management systems where each document is unique and evolves over time. Document databases enable flexible indexing, powerful ad hoc queries, and analytics over collections of documents.

Reference: <https://www.documentdb.com/sql/tutorial>

1. **Graph DB:**

**CAP Theorem :**

**C-Consistency:** A system will be consistence if Just after the write operation, for next read operation it provides correct results.

**A-Availability:** Every Request should get response, either success or failure weather you have high load or network failure, System should alwaays be available.

**and P-Partition tolerance:**  A System continue to work despite to message loss or partial failure. A partition tolerance System can sustain any number of network failure and does not result in entire network failure.

Means we will always have a backup server to keep the server up in case of any discrepancy.

As per this theorem, Any distributed system, we can achieve any of two property, we can’t achieve all 3 property in one System.

Exam : SQL Server , MYSQL , Oracle, PostgreSQL => Consistency, Availability

Cassandra , CouchDB, DynamoDB => Availability, Partition tolerance

MongoDB, Redis => Consistency, Partition tolerance

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Consistent Hashing**

**Different Hashing techniques:**

|  |  |  |
| --- | --- | --- |
| Hashing technique | Number of Bytes | Number of Hexadecimal Digits |
| **MD5** | 16 | 32 |
| SHA1 | 20 | 40 |
| SHA224 | 28 | 56 |
| SHA256 | 32 | 64 |
| SHA384 | 48 | 96 |
| SHA512 | 64 | 128 |
|  |  |  |

SHA : Secure Hash Algorithm

# **Challenges of Distributed Systems**

1. **Scheduling:**Sometimes selecting which job needs to run, when it should run and where it should run leads to under-utilized hardware and unpredictable runtimes.
2. **Latency:** As the system is widely distributed, there will be more latency that can occur between the communication of nodes.
3. **Security:** With many nodes, it is difficult to provide adequate security in distributed systems because all the nodes, as well as the connections, need to be secured.
4. **Losing data:** Sometimes, messages and data can get lost in the network while moving from one machine to another.

**KAFKA :**

https://betterprogramming.pub/thorough-introduction-to-apache-kafka-6fbf2989bbc1

1. Partitions are the main concurrency mechanism in Kafka. A topic is divided into 1 or more partitions, enabling producer and consumer loads to be scaled. Specifically, a consumer group supports as many consumers as partitions for a topic. The consumers are shared evenly across the partitions, allowing for the consumer load to be linearly scaled by increasing both consumers and partitions. You can have fewer consumers than partitions (in which case consumers get messages from multiple partitions), but if you have more consumers than partitions some of the consumers will be “starved” and not receive any messages until the number of consumers drops to (or below) the number of partitions. i.e. consumers don’t share partitions (unless they are in different consumer groups).

**KAFKA Introduction:**

-----------------------------------------------------------------------------------------

post -1 : refer:

[solutions-to-communication-problems-in-microservices-using-apache-kafka-and-kafka-lens](https://medium.com/@harmonh/solutions-to-communication-problems-in-microservices-using-apache-kafka-and-kafka-lens-9b6d453de352)

- Kafka takes bytes as an input and sends bytes as an output. That constraint-free protocol is what makes Kafka powerful.

- Kafka is a pure pub/sub. It takes bytes as an input and redistributes them without ever parsing them.

- kafka helps in micorservice architechture where service needs to communicate with each other very frequrently.

General way of communication is HTTP Rest call, but if one service get bombared with lots of request or data , this may slow down other services, which in turns can crash the complete system.

- Apache Kafka provides a simple solution for communication between microservices. Kafka implements a publisher-subscriber design model for communication.

Service can publish to kafka as producters into topics. Topics are generally container which separate the data into categories.

and Other downstream services can then subscribe to those topics as consumers and can then receive that information almost instantaneously.

Exam: Order service -> once order is made, order service can publish data to kafka topic exm: new Order, then paymentService, billingService, courier service, tracking service can subscribe to the new\_order topic and consume it information.

- Another problem in micorservice architechture with HTTP communication is, as new microservice comes into the system, API connections may need to remap and complexity will increase.

In case of Kafka for exm, let say new service as MarketingService needs to be added. then Only thing is required that MarketingService to become as consumer and subscribe to the topic.

- Kafka is extremely resilient and can scale with the needs of any application. Topics can be split into partitions and data can further be distributed across multiple Kafka brokers for redundancy and performance. When using Kafka in microservice architecture, services can process data that is queueing up in Kafka without worrying about other processes slowing down. Furthermore, if a service is down, producers can continue writing to topics as that data will be processed as soon as the failed service is restored.

- Implementing kafka can be challanging as there is no GUI for inspecting topics content or to view messages in partition. only CLI (command line interface) is available to view topico or partition messages.

One tool which helps in reading the messages from kafka topics and partitions is KAFKA LENS.

It allows user to view messages as they are passing from broker.It even allowing filtering of the messages by partitions.

- When migrating applications to Kafka, Kafka Lens is an invaluable tool that provides engineers a real-time view of messages as they are flowing through the broker in a user-friendly way. Kafka Lens can easily switch between topics and partitions allowing for engineers to quickly view the relevant messages (or lack thereof) to quickly diagnose and troubleshoot issues during implementation

----------------------------------------------------------------------------------------------

post-2:

refer : <https://sonusharma-mnnit.medium.com/apache-kafka-in-depth-49aae1e844be>

In the era of Big Data, lots and lots of data(volume) are being produced every second(velocity) from various sources like social media, blogs that I am writing currently, e-commerce, etc., which gets stored across different platforms in different schemas(varieties). In order to perform any ETL (Extract, Transform, Load) operation, a messaging/streaming system is needed which should be asynchronous and loosely coupled i.e. data from various sources/clients like hdfs, Cassandra, RDBMS, application log file, etc. could be dumped at a single place at the same time without all the clients depending on each other. One of the solutions to the problem is Kafka — An open-source distributed streaming platform created by LinkedIn and later donated to Apache.

Terminology

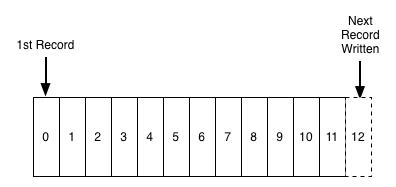
Messages: It is basically a key-value pair contains useful data/record in the value section.

Topic: For multi-tenancy, multiple topics can be created which is just a feed name to which messages are published and subscribed.

Offset: Messages are stored in a sequential form similar to commit log and a sequential id is provided to each message starting from 0.

Messages in a queue

Broker: Kafka cluster consists of brokers which are just nodes in the cluster hosting stateless server maintained by a zookeeper. Since there is no master-slave concept here, all brokers are peers. Let's understand zookeeper first before proceeding further.



Messages in a queue

Need for Zookeeper: Zookeeper is the system for the distributed cluster management. It is distributed key value store.

It is used in kafka for below reasons:

1. **Broker controller Election:**

All the reads and write to the partition of a topic happens through replica of a leader. If leader goes down, A new leader is elected by Zookeeper.

It manages the brokers which hosts stateless server.

1. **Configuration of topics:**

Meta data related to the topic that weather a particular topic is sitting in the broker, how many partitions are there etc are stored at zookeeper end and in continuously sync whenever a new message is produced.

1. Access Control list is maintained at zookeeper level.

**Why Kafka:**

Key features better than other messaging queues are as follows,

1. High throughput:

Throughput: rate at which Kafka can process the message stream or data per second.

In Kafka, as we can partitioned the topic which can spread across the multiple brokers, we can achieve thousands of read and write per seconds.

1. Distributed:

A distributed system is one which is splits into multiple running machines and all these machines works together in a cluster to server the end user.

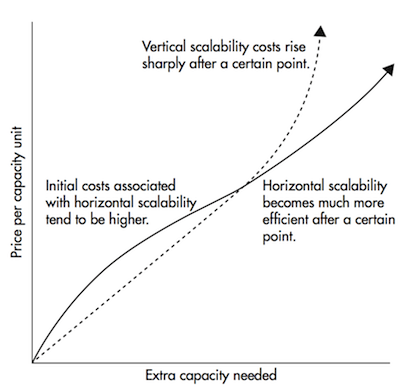
Kafka is distributed as it stores, read and write the data/message on the several nodes called broker, which along with the Zookeeper created the ecosystem called Kafka cluster.

1. Persistence:

Message queue is completely maintained on the disk instead of keeping it in memory and copy of the same data can be stored across multiple nodes. So there is no chance of data loss due to failure scenarios.

1. Scalability:

Kafka scales horizontally means we can easily add new Node/Broker whenever we run out of the capacity.



Horizontal scaling becomes much cheaper after a certain threshold

1. Fault Tolerance:

If we have n topics with each has m partitions, then we total m\*n partition has to replicated on Q brokers If replication factor is Q. In this way we are making system to fault tolerant upto Q-1 node/broker failure/.

Zookeeper will be up and running on 2181

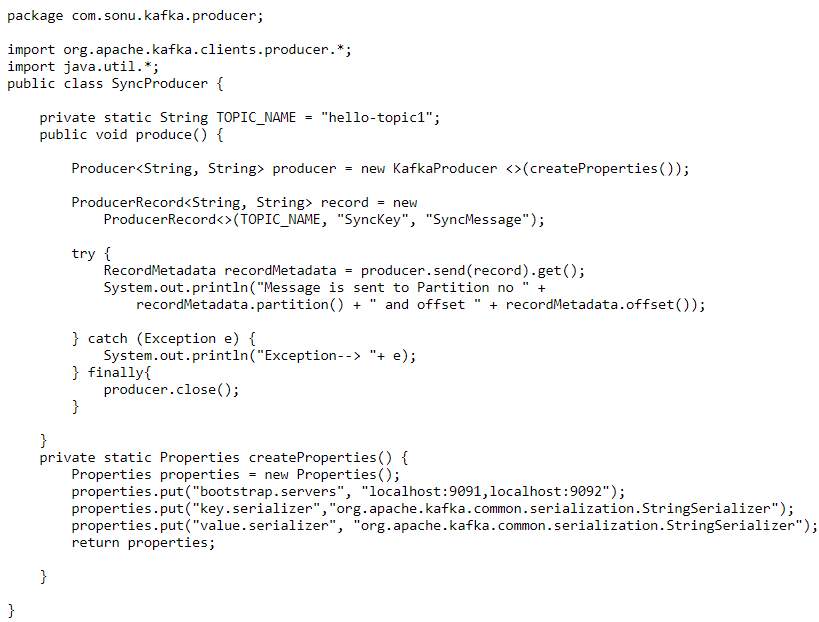
**Producer API:**

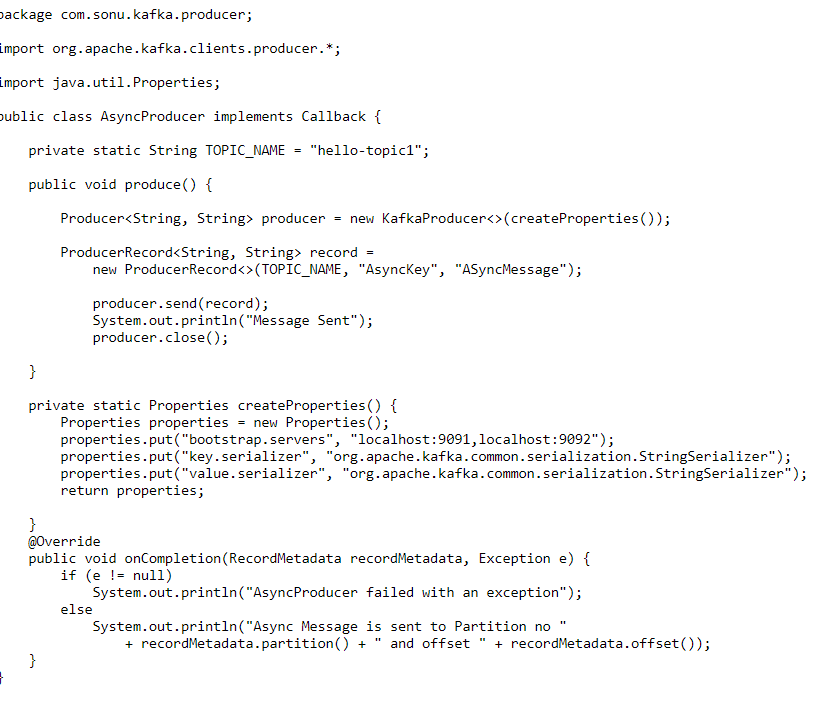
Kafka producer is source which publish the message to the topic using API provided by Kafka.

A properties object containing the configuration information need to be set before sending/publishing the message.

Main classes Producer, Kafka Producer, Producer Record etc.

**Sample API:**





**How consumer retries a message:**

The consumer also gets metadata first and reads the message from the leader partition. Kafka is very fast and can get real-time message, but a single consume will certainly will have some latency in reading big chunk of messages form topic. This lag is called consumer lag.

To overcome this problem, A consumer group can be created which will have a common group id. Each consumer connects to the different partition divided equally among all the consumer.

*The assignment of the partition to a particular consumer is the responsibility of****Group Coordinator — One of the brokers in the cluster is nominated for this role****. In order to manage the list of active consumers, all the consumer of a group sends their heartbeat to the group coordinator.*The number of consumers in a group should be less than or equal to the number of partitions in that particular topic, violating the condition will end up in a situation where a consumer sits idle.

Creating a simple Consumer:  
**bin/kafka-console-consumer.sh --bootstrap-server localhost:9091 --topic topic1**

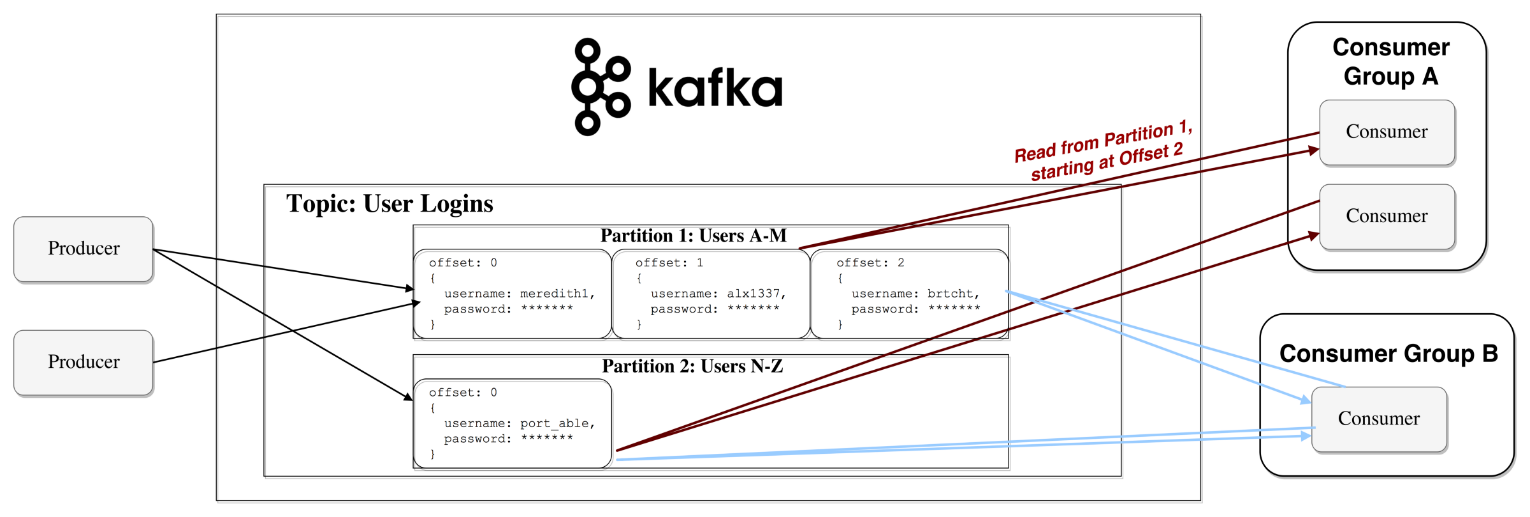
Creating consumer to read from particular offset and partition:  
**bin/kafka-console-consumer.sh --bootstrap-server localhost:9091 --topic topic1 --offset 0 --partition 1**

Creating a consumer group:  
**bin/kafka-console-consumer.sh --bootstrap-server localhost:9091 --topic topic1 --group groupName**

Multiple process/consumers of the above groupName can be created which is called Consumer Group.

More than one consumer can read a single topic at the same time. Now, in order to remember till which offset the particular read, storage known as **consumer offset**

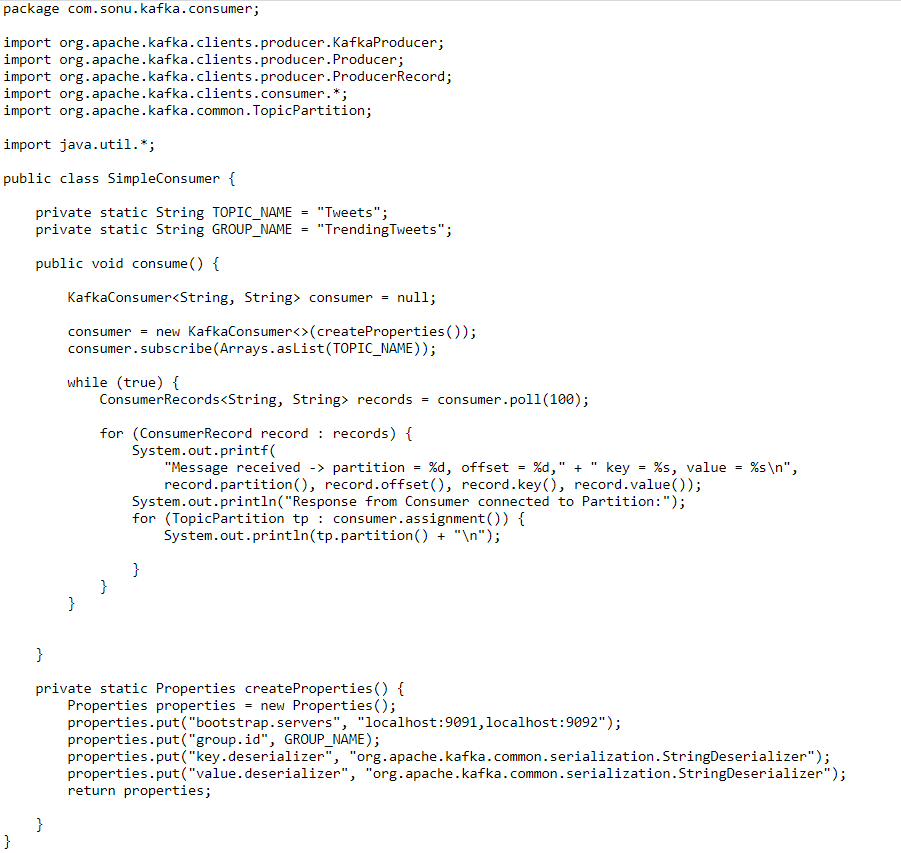
The Consumer offset has the key as — > [Group Id, Topic, Partition] and value — > [Offset, …]



## Consumer API:

Similar to producer API, Kafka provides classes to connect to the bootstrap servers and get the messages. Deserializer needs to be written when passing a message of other than standard data types.

Sample Code



**Why Kafka is so fast:**

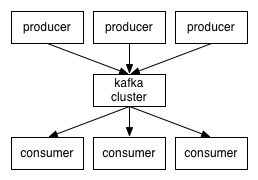
1. No Random Disk Access: It uses a sequential data structure known as an immutable queue where read and write operation is always constant time O(1) (as it is always append data structure and delete/remove is not allow that is why read and write o(1)). It appends the message at the end and read from the beginning or from a particular offset.
2. Sequential I/O: Modern operating systems allocate most of their free memory to disk-caching and are faster for storing and retrieving sequential data.
3. Zero Copy: The data from disk is unnecessarily loaded into the application memory as it is not being modified at all. So, instead of loading it to the application, it sends the same data from the kernel context buffer over the socket, NIC buffer and to the network.
4. Batching of messages: Several messages are grouped together in order to avoid the multiple network call.
5. Message Compression: Before transferring the message over the wire, it is compressed using compression algorithm like gzip, snappy, etc. and decompressed at the consumer API layer.

**Part-3 :**

**Refer :** [**https://betterprogramming.pub/thorough-introduction-to-apache-kafka-6fbf2989bbc1**](https://betterprogramming.pub/thorough-introduction-to-apache-kafka-6fbf2989bbc1)

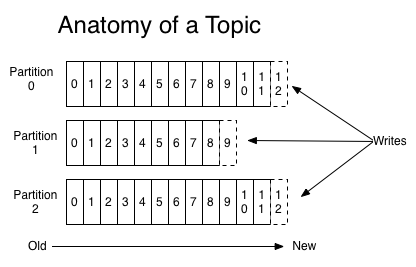
# **How Does It Work?**

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.



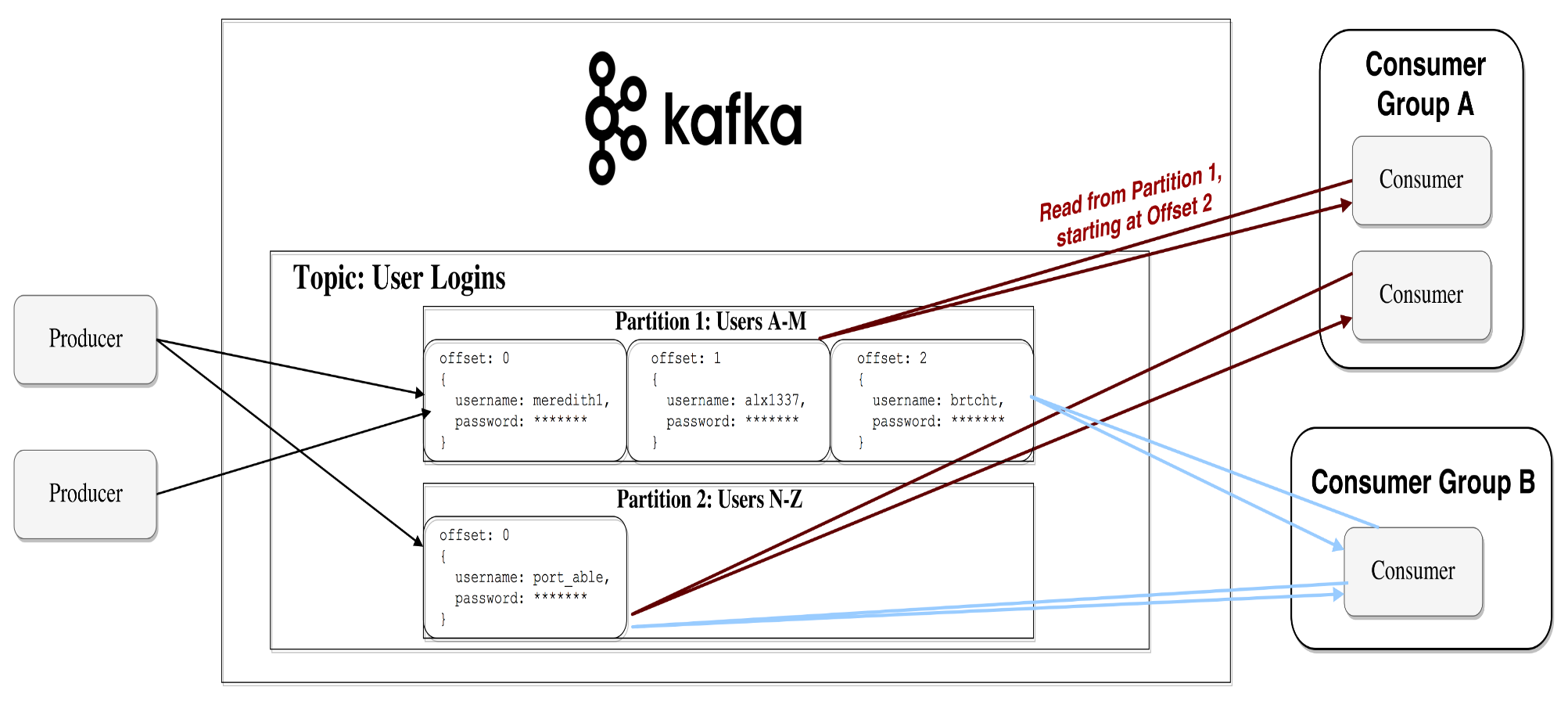
As topics can get quite big, they get split intopartitions of a smaller size for better performance and scalability. (For example, say you were storing user login requests. You could split them by the first character of the user’s username.)

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 doesn’t keep track of what records are read by the consumer, then deleting them. Rather, it stores them for a set amount of time (e.g., one day-> called TTL time to live) 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’s worth noting consumers are actually consumer groups that 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 doesn’t 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:

* Kafka has a protocol that groups messages together. This allows network requests to group messages together and reduce network overhead; the server, in turn, persists chunk of messages in one go, and consumers fetch large linear chunks at once.
* Linear reads/writes on a disk are fast. The concept that modern disks are slow is because of numerous disk seeks, something that’s not an issue in big linear operations.
* 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.
* Modern OSes cache the disk in free RAM. This is called pagecache.
* 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’s 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 and Replication**

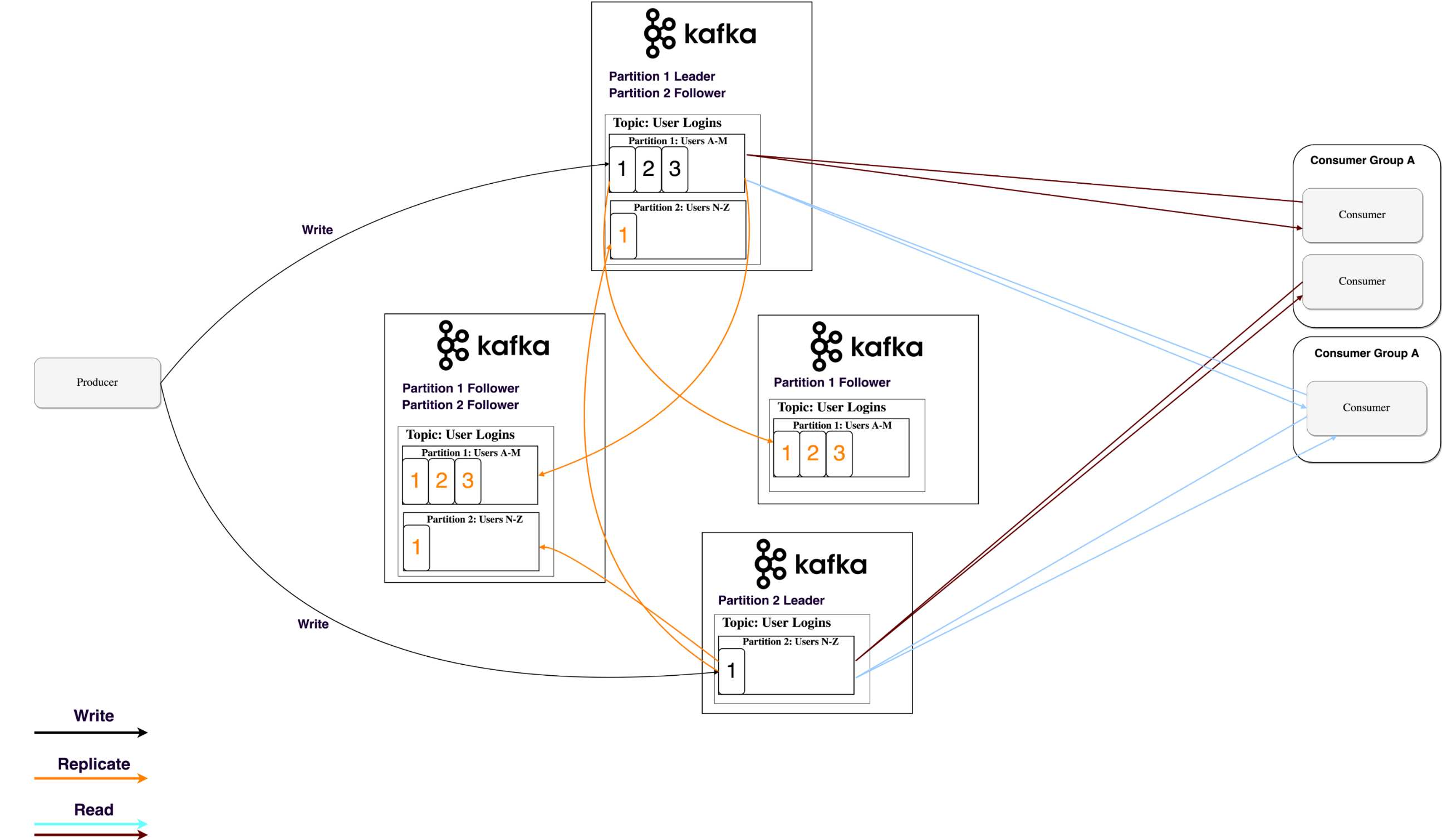
Let’s talk about how Kafka achieves fault tolerance and how it distributes data between nodes.

## 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 won’t 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.



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’s Zookeeper?****

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

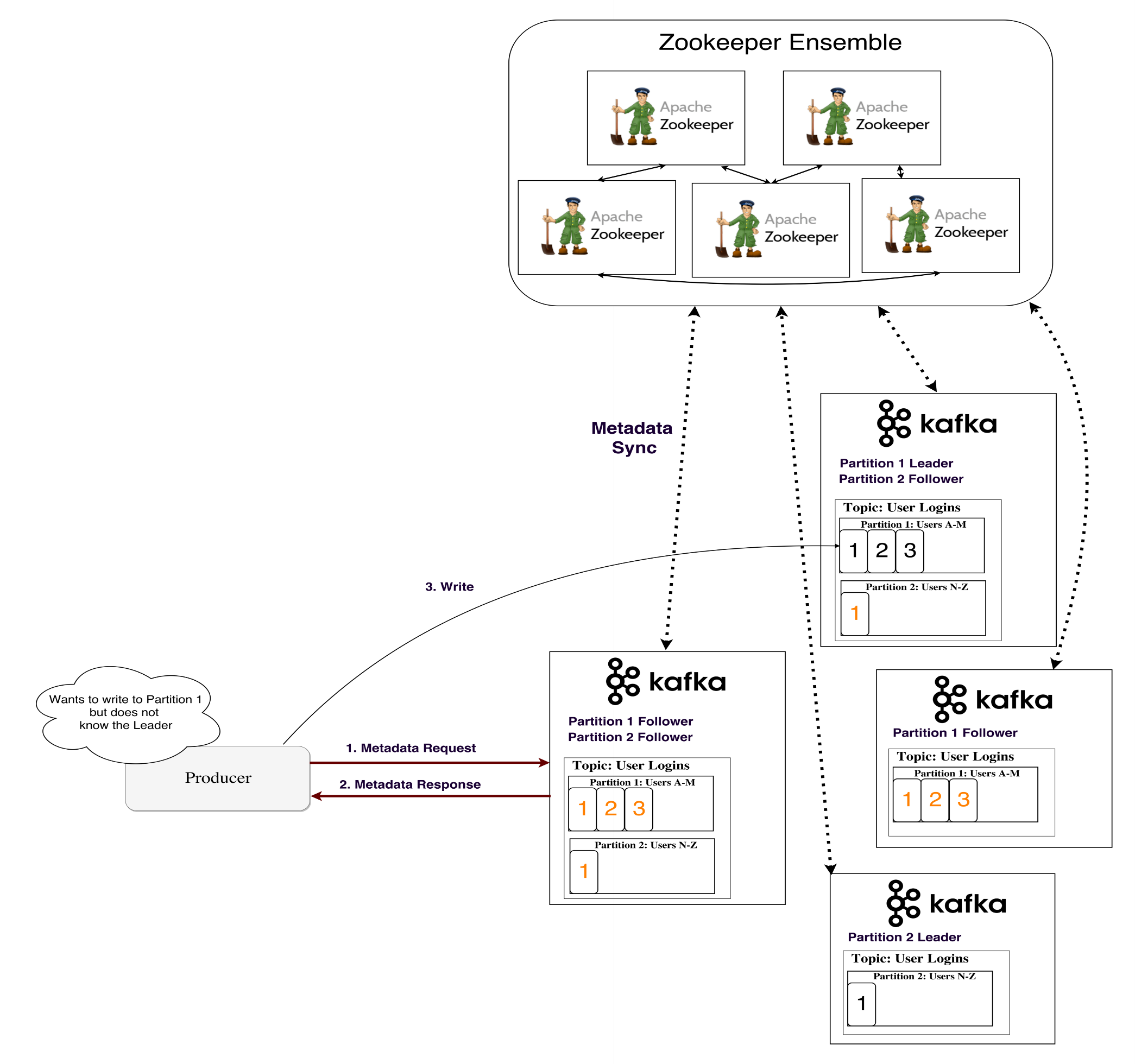
It allows clients of the service (the Kafka brokers) to subscribe and have changes sent to them once they happen. This is how brokers know when to switch partition leaders. Zookeeper is also extremely fault tolerant, and it ought to be, as Kafka heavily depends on it.

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

* Consumer groups offset per partition (although modern clients store offsets in a separate Kafka topic)
* Access control lists (ACLs) — used for limiting access/authorization
* Producer and consumer quotas —maximum message/sec boundaries
* Partition leaders and their health

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

Producers 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**

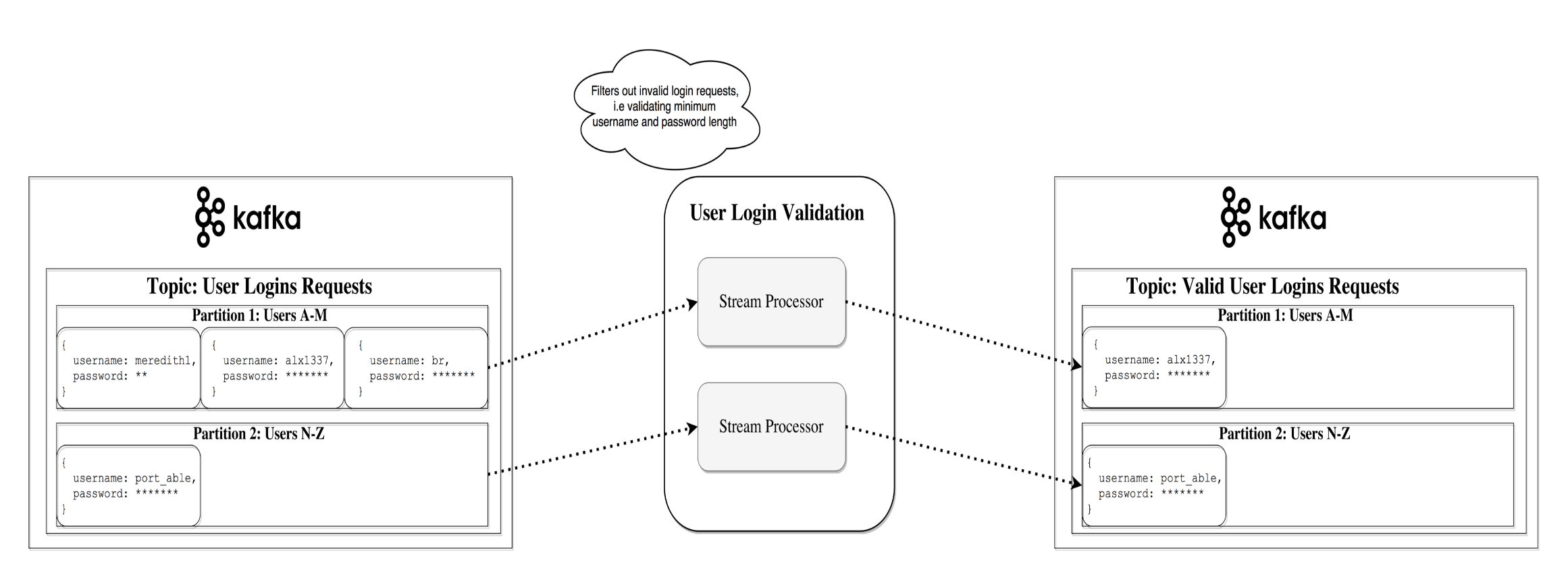
In Kafka, a stream processor is anything that takes continual streams of data from input topics, performs some processing on this input, and produces a stream of data to output topics (or external services, databases, the trash bin — wherever really).

It’s 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](https://www.google.bg/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiG1fu-oYDYAhVCCewKHeZMDfEQFggsMAA&url=https%3A%2F%2Fkafka.apache.org%2Fdocumentation%2Fstreams%2F&usg=AOvVaw30e_Zle1rMLJugJuCTx9tx) library.

This API is intended to be used within your own codebase — it’s 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 doesn’t depend on anything external. You know that for any given data, you’ll 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

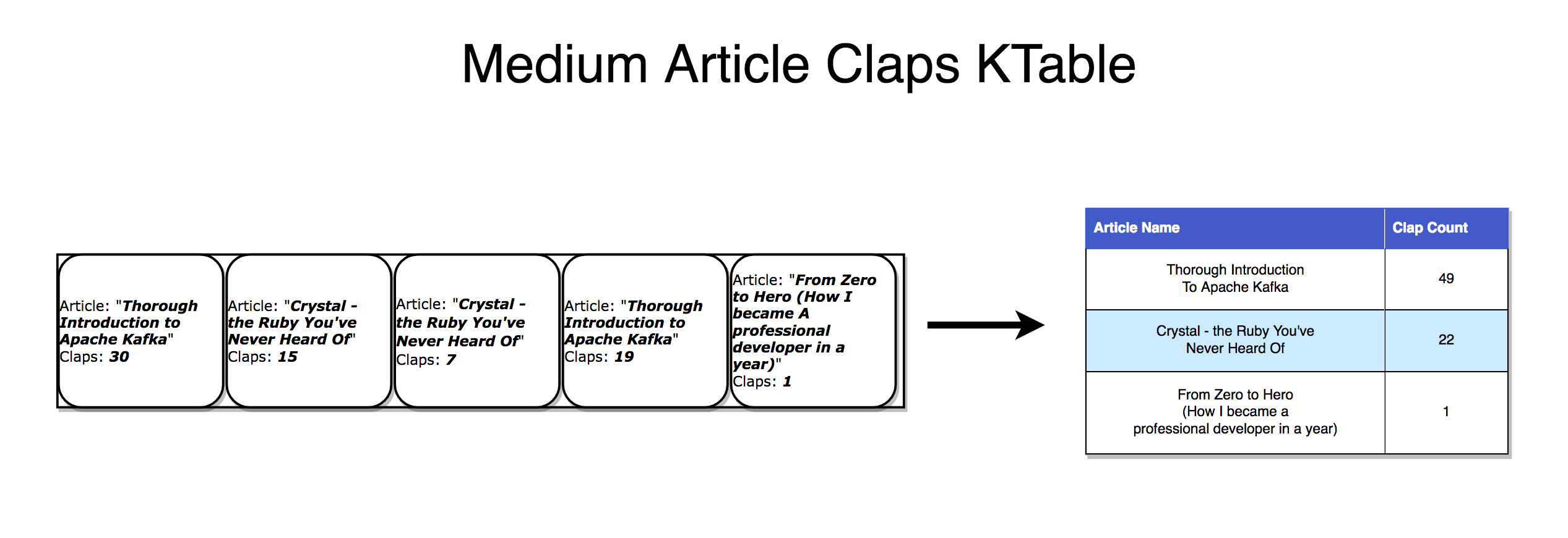
It’s important to recognize that streams and tables are essentially the same. A stream can be interpreted as a table, and a table can be interpreted as a stream.

## 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](https://martinfowler.com/eaaDev/EventSourcing.html).

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

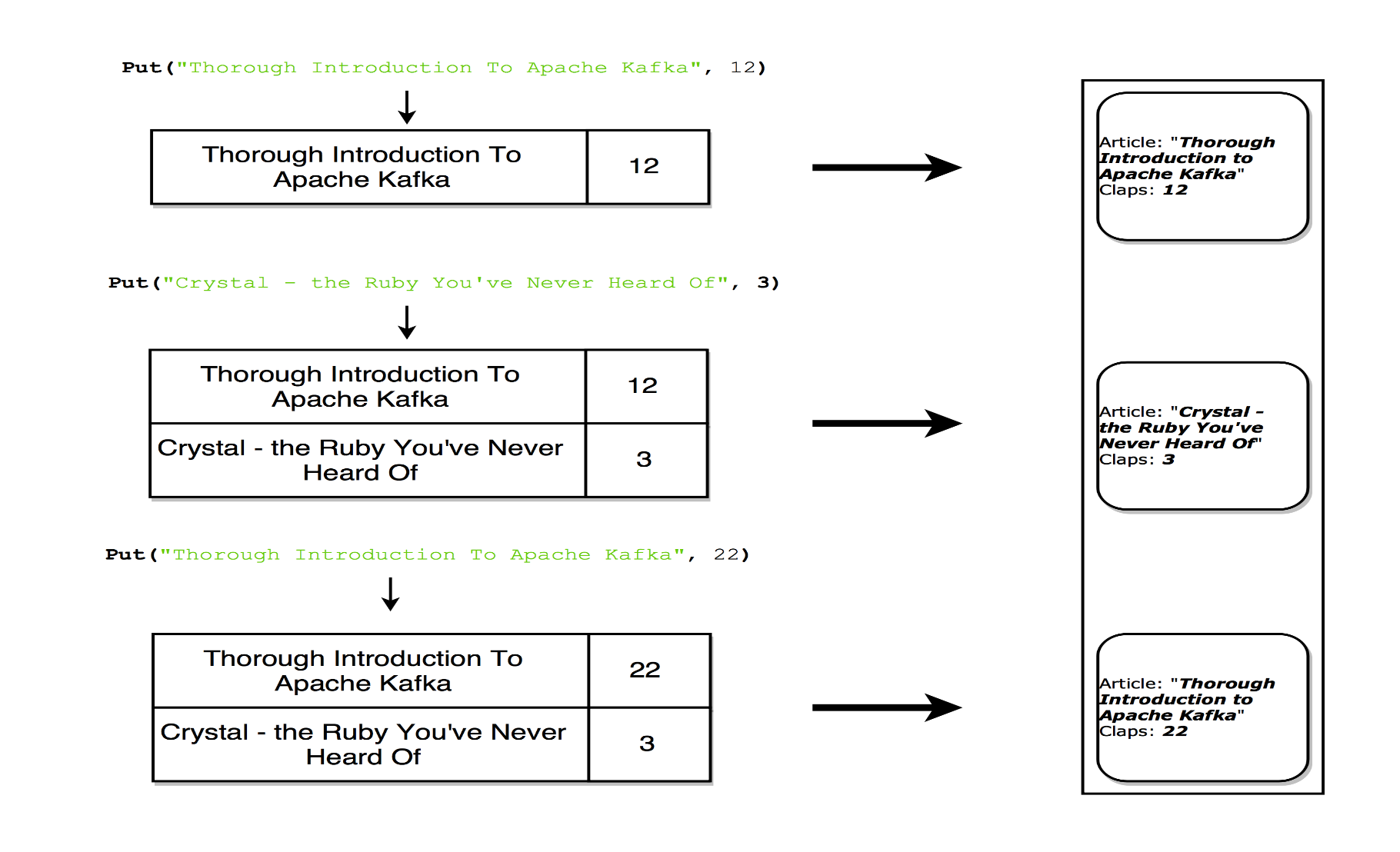
A Kafka stream can be interpreted in the same way —events which, when accumulated, form the final state. Such stream aggregations get saved in a local [RocksDB](https://github.com/facebook/rocksdb/wiki/rocksdb-basics) (by default) and are called a KTable.



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 that 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 don’t 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 the state on stream processors is the stream processors can fail! Where would you need to keep this state in order to make it fault-tolerant?

A naive approach is to simply store all states in a remote database and join over the network to that store. The problem with this is there’s no locality of data and lots of network round trips, both of which will significantly slow down your application.

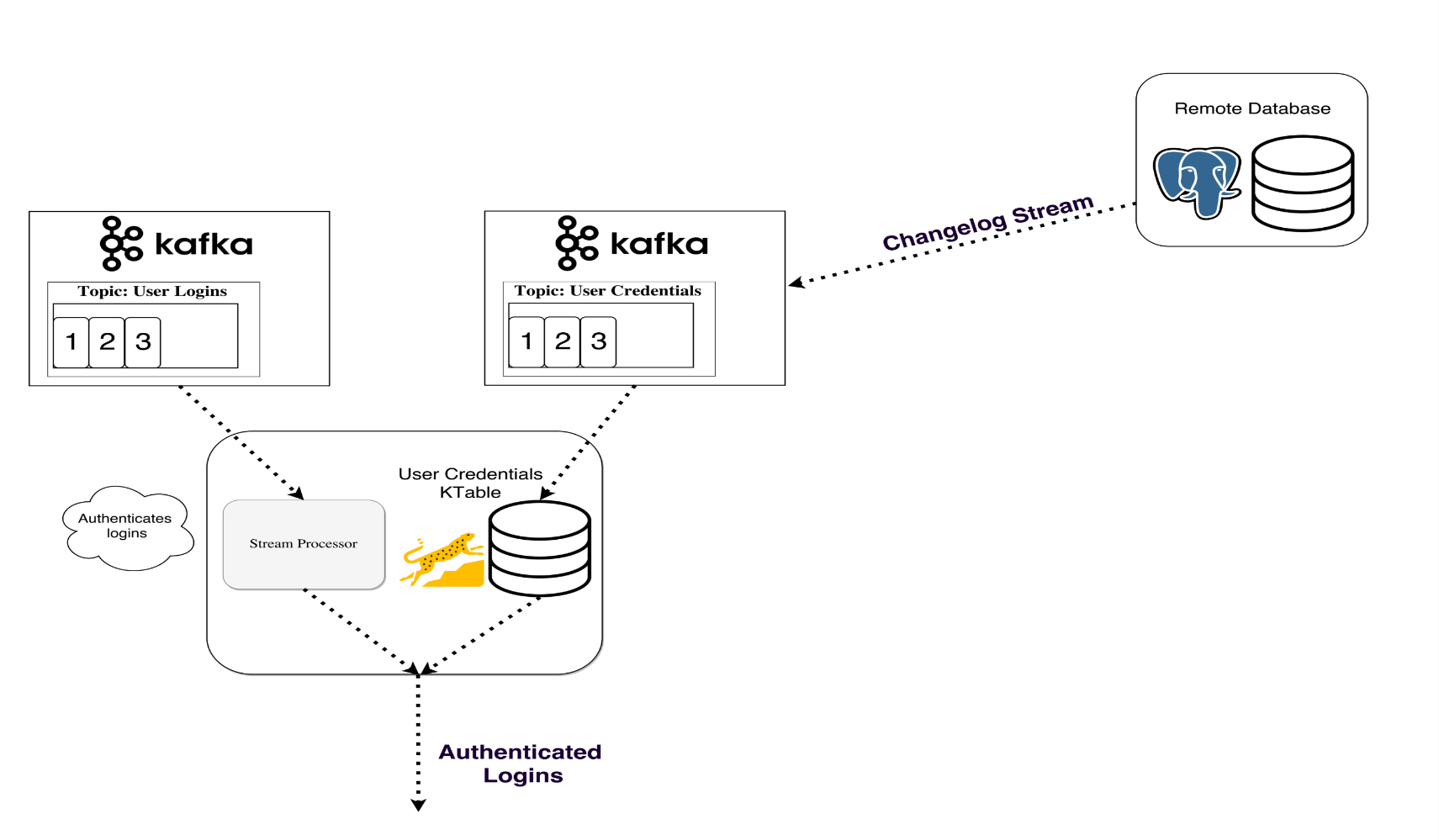
A more subtle but important problem is your stream processing job’s uptime would be tightly coupled to the remote database and the job won’tt be self-contained (a change in the database from another team might break your processing).

So what’s a better approach?

Recall the duality of tables and streams. This allows us to convert streams into tables that are colocated 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.

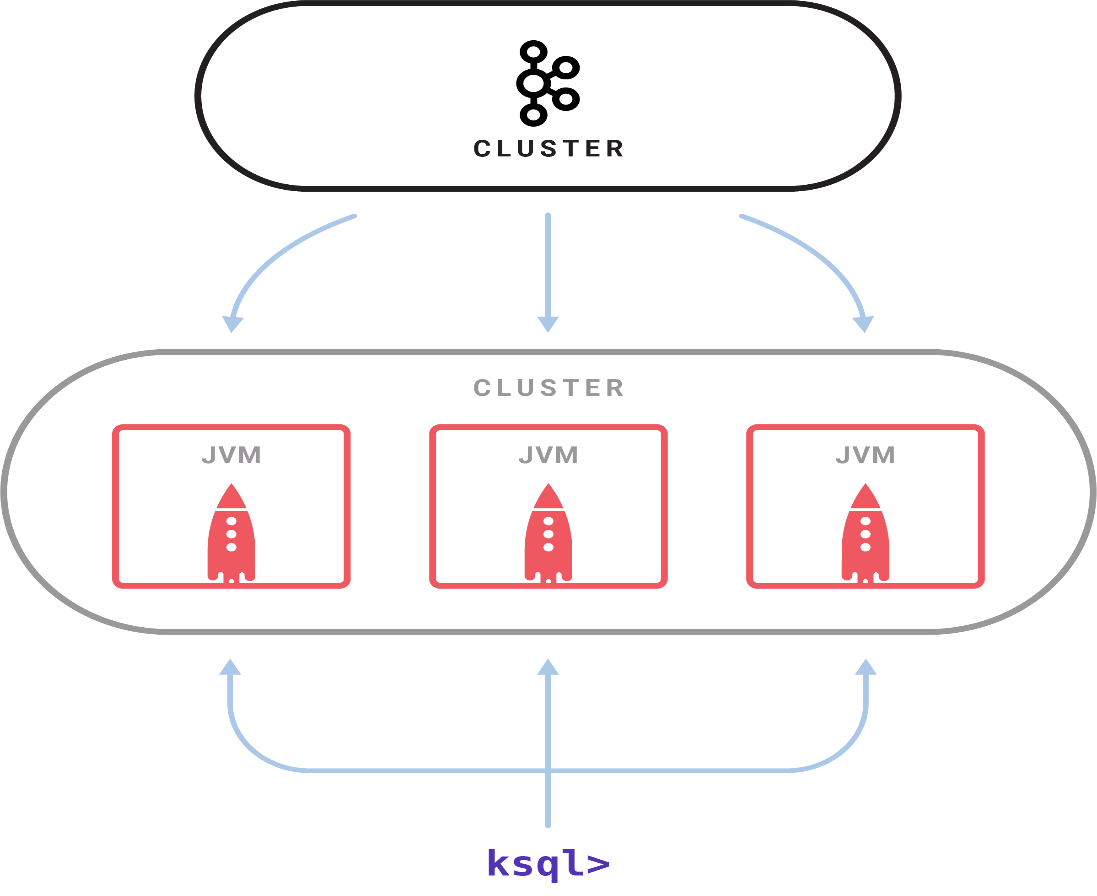
You could even have a remote database be the producer of the stream, effectively broadcasting a changelog with which you rebuild the table locally.



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’s where the only official Kafka Streams API client is.



Sample KSQL setup

[Released in April 2018](https://www.confluent.io/blog/confluent-platform-4-1-with-production-ready-ksql-now-available/), [KSQL](https://www.confluent.io/blog/ksql-open-source-streaming-sql-for-apache-kafka/) is a feature that allows you to write your simple streaming jobs in a familiar SQL-like language.

You set up a KSQL server and interactively query it through a [CLI](https://en.wikipedia.org/wiki/Command-line_interface) to manage the processing. It works with the same abstractions (KStream and KTable), guarantees the same benefits of the Streams API (scalability, fault tolerance), and greatly simplifies work with streams.

This might not sound like a lot, but in practice, it’s 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](https://www.youtube.com/watch?v=A45uRzJiv7I&t=2m13s).

## Streaming alternatives

Kafka streams are a perfect mix of power and simplicity. They arguably have the best capabilities for stream jobs on the market, and they integrate with Kafka way easier than other stream-processing alternatives ([Storm](https://storm.apache.org/), [Samza](https://samza.apache.org/), [Spark](https://spark.apache.org/), [Wallaroo](https://github.com/WallarooLabs/wallaroo)).

The problem with most other stream-processing frameworks is they’re 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’ll execute it (along with the 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.

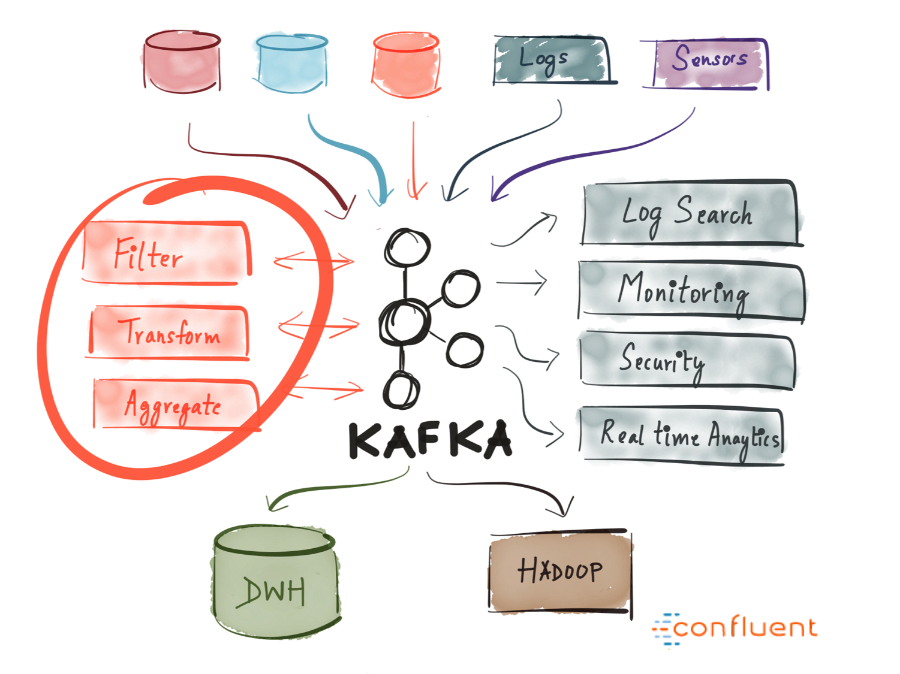
Kafka Streams let you roll out your own deployment strategy when you need it, be it [Kubernetes](https://kubernetes.io/), [Mesos](https://mesos.apache.org/), [Nomad](https://www.nomadproject.io/), [Docker Swarm,](https://github.com/docker/swarm) or others.

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’s 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 to store them without worrying about things like performance or data loss.

This means it’s 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’s now easier than ever to write business logic that 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 aren’t strong enough reasons for a company to adopt a new technology. There are other systems that 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 because of 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’d 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**

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](https://www.confluent.io/blog/apache-kafka-goes-1-0/) (November 1, 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 this introduction helped familiarize you with Apache Kafka and its potential.

Apache Kafka is a distributed streaming platform capable of handling trillions of events a day. Kafka provides low-

**Kafka Important config:**

# 10 Configs to Make Your Kafka Producer More Resilient

# Kafka is well known for its resiliency, fault-tolerance, and high throughput. But its performance doesn’t always meet everyone’s expectations. In some cases, we can improve it by scaling out or scaling up brokers. While in most cases, we have to play the game of configurations.

There are really [tons of configurations](https://kafka.apache.org/documentation/#configuration) in Kafka ecosystem. It’s nearly impossible to grasp the idea of every single configuration. On one hand, they definitely make the system more flexible, but on the other hand, developers can feel quite confused about what to do with them.

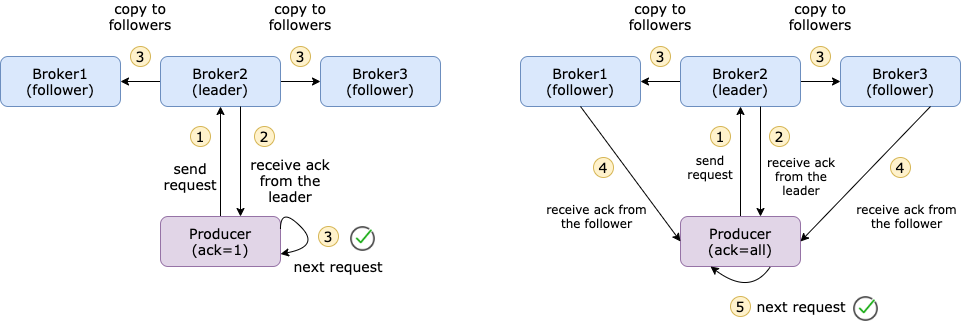
Fortunately, the majority of configurations are already pre-defined in a way that they work well for most situations. For starters, the mandatory configurations they need to know are very limited.

But of course, I assume you are reading this article because you want to bring your Kafka producer to the next level. So in this article, I want to share 10 configs that I think are important to make your producer more resilient.

The configs that will be discussed in this article are acks, replica.lag.time.max.ms, min.insync.replicas, retries, enable.idempotent, max.in.flight.requests.per.connection, buffer.memory, max.block.ms, linger.ms, batch.size.

## Acks (acknowledgments)

An ack is an acknowledgment that the producer gets from a Kafka broker to ensure that the message has been successfully committed to that broker. The config acks is the number of acknowledgments the producer needs to receive before considering a successful commit.



Difference between ack=1 and ack=all, Created by

[Xiaoxu Gao](https://medium.com/u/2adc5a07e772?source=post_page-----ec6903c63e3f--------------------------------)

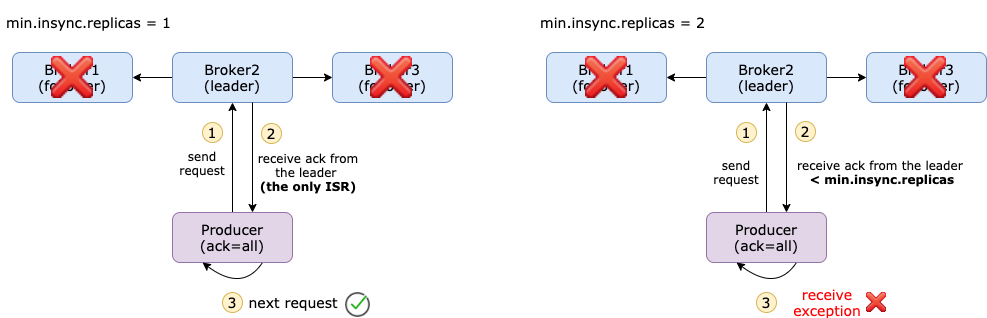
The default value is 1, which means as long as the producer receives an ack from the leader broker of that topic, it would take it as a successful commit and continue with the next message. It’s not recommended to set acks=0, because then you don’t get any guarantee on the commit. acks=all would make sure that the producer gets acks from all the in-sync replicas of this topic. It gives the strongest message durability, but it also takes long time which results in higher latency. So, you need to decide what is more important for you.

## In-sync replicas

acks=all would get acknowledgments from all the in-sync replicas (ISR), so what is an in-sync replica? When you create a topic, you must define how many replicas you want. A replica is nothing more than a copy of the message in one of the brokers, so the maximum number of replicas is the number of brokers.

Among these replicas, there is a leader and the rest are followers. The leader handles all the read and write requests while the followers passively replicate the leader. **An in-sync replica is a replica that fully catches up with the leader in the last 10 seconds.** The time period can be configured via replica.lag.time.max.ms. If a broker goes down or has network issues, then it couldn’t follow up with the leader and after 10 seconds, this broker will be removed from ISR.

The default minimum in-sync replica ( min.insync.replicas) is 1. It means that if all the followers go down, then ISR only consists of the leader. Even if acks is set to all, it actually only commits the message to 1 broker (the leader) which makes the message vulnerable.



Difference between different min.insync.replicas Created by

The config min.insync.replicas basically defines how many replicas that the producer must receive before considering a successful commit. This config adds on top of acks=all and makes your messages safer. But on the other hand, you have to balance latency and speed.

## Retry on failure

Let’s say you set acks=all and min.insync.replicas=2. For some reason, the follower goes done, then the producer identifies a failed commit because it couldn’t get acks from min.insync.replicas brokers.

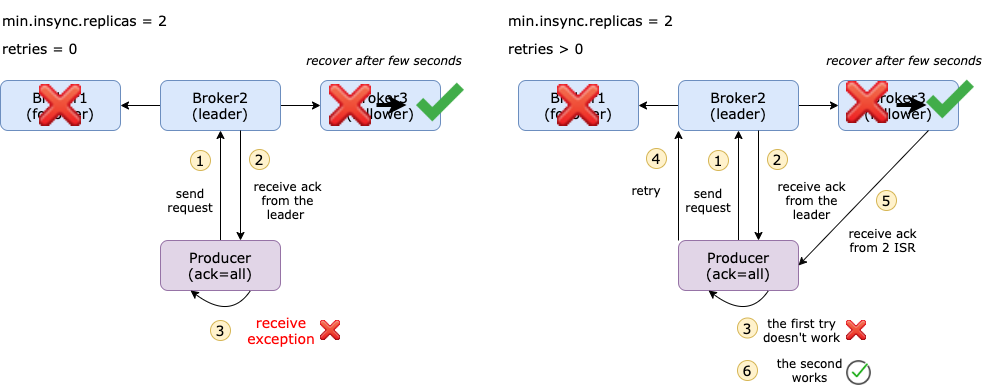
You will get an error message from the producer:

KafkaError{code=NOT\_ENOUGH\_REPLICAS,val=19,str="Broker: Not enough in-sync replicas"}

You will see the following error message from the running broker. It means that even if the broker is running, Kafka will not append the messages to that running broker if the current ISR is not sufficient.

ERROR [ReplicaManager broker=0] Error processing append operation on partition test-2-2-0 (kafka.server.ReplicaManager)  
org.apache.kafka.common.errors.NotEnoughReplicasException: The size of the current ISR Set(0) is insufficient to satisfy the min.isr requirement of 2 for partition test-2-2-0

By default, the producer will not act upon this error, so it will lose messages. This is called **at-most-once semantics**. But you can let the producer resend messages by configuring retries=n. This is basically the maximum number of retries the producer would do if the commit fails. The default value is 0.



Difference between retries=0 and retries>0 Created by

If you set retries=5, then the producer will retry maximum 5 times. You will not notice the number of retries from the producer log because it only shows if the commit is successful or not in the end. But you can see retries+1 log messages on the broker side.

## Avoid duplicated messages

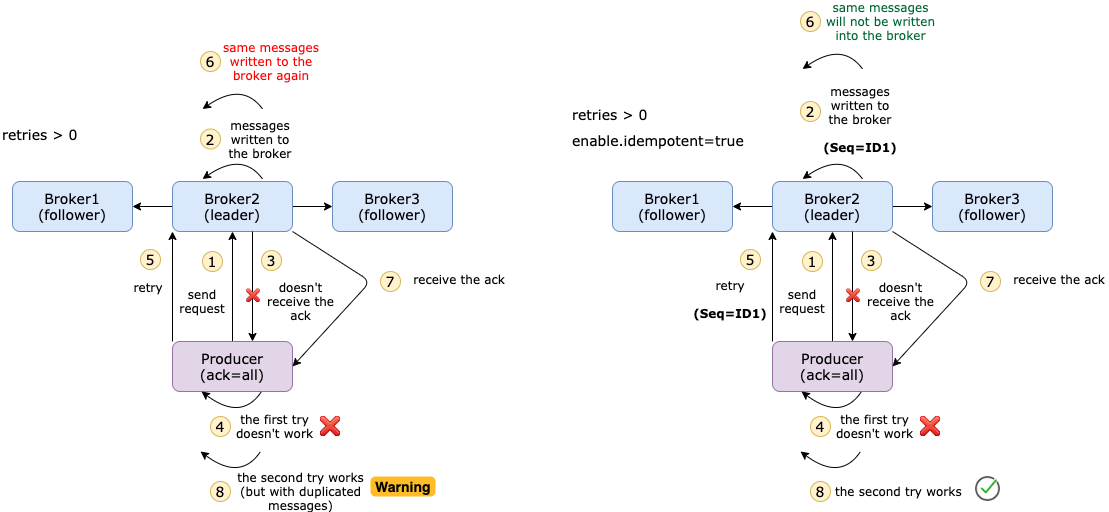
In some situations where a message has actually been committed to all in-sync replicas, but the broker couldn’t send an ack back due to a network issue (e.g. only allows one-way communication). Meanwhile, we set retries=3, then the producer will resend the messages 3 times. This could lead to duplicated messages in the topic.

Let’s say we have a producer that sends 1M messages to the topic and the broker fails after the messages have been committed but before the producer receives all the acks. In this case, we will probably end up with more than 1M messages on the topic. This is also called **at-lease-once semantics**.

The most ideal situation is **exactly-once semantics** where even if the producer resends the message, the consumer should receive the same message only once.

What we need is an **Idempotent Producer.** [Idempotent](https://stackoverflow.com/questions/1077412/what-is-an-idempotent-operation#:~:text=Idempotence%20means%20that%20applying%20an,the%20result%20is%20still%20zero.) means that applying an operation once or applying it multiple times have the same effect. It’s very easy to turn this feature on with config enable.idempotent=true.

How does it work? Messages are sent in batches, and each batch has a sequence number. On the broker side, it keeps track of the largest sequence number for each partition. If a batch with a smaller or equal sequence number comes in, the broker will not write that batch into the topic. In this way, it also ensures the order of the batches.



Difference between disabling idempotent and enabling idempotent Created by

## Send messages in order

Another important config to ensure the order is max.in.flight.requests.per.connection, and the default value is 5. This represents the number of unacknowledged requests that can be buffered on the producer side. If the retries is greater than 1 and the first request fails, but the second request succeeds, then the first request will be resent and messages will be in the wrong order.

According to the [documentation](http://kafka.apache.org/documentation/#max.in.flight.requests.per.connection):

*Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).*

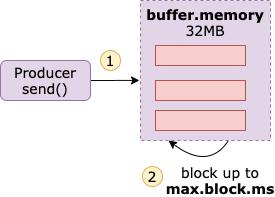
If you don’t enable idempotent, but still want to keep messages in order, then you should config this setting to 1.

But if you’ve already enabled idempotent, then you don’t need to explicitly define this config. Kafka will choose suitable values, as stated [here](http://kafka.apache.org/documentation/#enable.idempotence).

*If these values are not explicitly set by the user, suitable values will be chosen. If incompatible values are set, a ConfigException will be thrown.*

## Sending messages too fast

When the producer calls send(), the messages will not be immediately sent but added to an internal buffer. The default buffer.memory is 32MB. If the producer sends messages faster than they can be transmitted to the broker or there is a network issue, it will exceeds buffer.memory then the send() call will be blocked up to max.block.ms (default 1 minute).



buffer.memory and max.block.ms Created by

[Xiaoxu Gao](https://medium.com/u/2adc5a07e772?source=post_page-----ec6903c63e3f--------------------------------)

This problem can be mitigated by increasing both values.

Another 2 configs that you can play around with are linger.ms and batch.size. linger.ms is the delay time before the batches are ready to be sent. The default value is 0 which means batches will be immediately sent even if there is only 1 message in the batch. Sometimes, people increase linger.ms to reduce the number of requests and improve throughput. But this will lead to more messages kept in memory. So, make sure that you take care of both sides.

There is an equivalent configuration as linger.ms, which is batch.size. This is the maximum size of a single batch. Batches will be sent when any of these 2 requirements are fulfilled.