P5

What is event streaming?

- Event streaming is the digital equivalent of the human body's central nervous system.
- It is the technological foundation for the 'always-on' world where businesses are increasingly software-defined and automated, and where the user of software is more software.
- Technically speaking, event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

P6

What can I use event streaming for?

- To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.
- To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.

- To connect, store, and make available data produced by different divisions of a company.
- To serve as the foundation for data platforms, event-driven architectures, and microservices.

P7

The New York Times

The New York Times uses Apache Kafka and the Kafka Streams API to store and distribute, in real-time, published content to the various applications and systems that make it available to the readers.

Zalando

As the leading online fashion retailer in Europe, Zalando uses Kafka as an ESB (Enterprise Service Bus), which helps us in transitioning from a monolithic to a micro services architecture. Using Kafka for processing event streams enables our technical team to do near-real time business intelligence.

Line

LINE uses Apache Kafka as a central datahub for our services to communicate to one another. Hundreds of billions of messages are produced daily and are used to execute various business logic, threat detection, search indexing and data analysis. LINE leverages Kafka Streams to reliably transform and filter topics enabling sub topics consumers can efficiently consume, meanwhile retaining easy maintainability thanks to its sophisticated yet minimal code base.

Robobank

Rabobank is one of the 3 largest banks in the Netherlands. Its digital nervous system, the Business Event Bus, is powered by Apache Kafka. It is used by an increasing amount of financial processes and services, one which is Rabo Alerts. This service alerts customers in real-time upon financial events and is built using Kafka Streams.

Adidas

adidas uses Kafka as the core of Fast Data Streaming Platform, integrating source systems and enabling teams to implement real-time event processing for monitoring, analytics and reporting solutions.

Airbnb

Used in our event pipeline, exception tracking & more to come.

Aiven

Aiven is a cloud platform for open source technologies. We provide Apache Kafka as a managed service on public clouds, and use it internally to run and monitor our platform of tens of thousands of clusters.

Coursera

At Coursera, Kafka powers education at scale, serving as the data pipeline for realtime learning analytics/dashboards.

Cisco

Cisco is using Kafka as part of their OpenSOC (Security Operations Center). More details here.

DataDog

Kafka brokers data to most systems in our metrics and events ingestion pipeline. Different modules contribute and consume data from it, for streaming CEP (homegrown), persistence (at different "atemperatures"a in Redis, ElasticSearch, Cassandra, S3), or batch analysis (Hadoop).

HackerRank

HackerRank uses Kafka as events as a service platform. We publish all the internal activity on our infrastructure into Kafka, and a wide range of internal services subscribe to it.

Linkedin

Apache Kafka is used at LinkedIn for activity stream data and operational metrics. This powers various products like LinkedIn Newsfeed, LinkedIn Today in addition to our offline analytics systems like Hadoop.

Firefox

Kafka will soon be replacing part of our current production system to collect performance and usage data from the end-users browser for projects like Telemetry, Test Pilot, etc. Downstream consumers usually persist to either HDFS or HBase.

Netflix

Real-time monitoring and event-processing pipeline.

Oracle

Oracle provides native connectivity to Kafka from its Enterprise Service Bus product called OSB (Oracle Service Bus) which allows developers to leverage OSB built-in mediation capabilities to implement staged data pipelines.

Oracle GoldenGate

GoldenGate offers a comprehensive solution that streams transactional data from various sources into various big data targets including Kafka in real-time, enabling organizations to build fault -tolerant, highly reliable, and extensible analytical applications.

Spotify

Kafka is used at Spotify as part of their log delivery system.

Paypal

Kafka is playing an increasingly important role in messaging and streaming systems. Managing fast-growing Kafka deployments and supporting customers with various requirements can become a challenging task for a small team of only a few engineers.

The availability of the Kafka infrastructure is essential to PayPal's revenue stream. The company needs to catch issues before systems break down, know exactly how available it is for each client, and preemptively recover from problems when they occur. It also needs to have a clear view of message loss in its end-to-end Kafka pipeline. Operational tooling is critical to PayPal's success, and the company has developed tools such as data loss auditing, full and partial cluster failovers, client and server-side KPI measurements, and a control panel for Kafka clusters.

Kevin Lu, Maulin Vasavada, and Na Yang explore the management and monitoring PayPal applies to Kafka, from client-perceived statistics to configuration management, failover, and data loss auditing. You'll discover the criticality of a large-scale Kafka environment, the set of tools you'll need to make this environment work and supportable, and how to provide the performance and scalability needed for PayPal's data volume and SLA. Along the way, they highlight the architecture of PayPal's next-generation Kafka monitoring and management system, built for serving all the Kafka-as-a-service needs.

Twitter

As part of their Storm stream processing infrastructure, e.g. this and this.

Microsoft

Processing trillions of events per day with Apache Kafka on Azure

P8

How can Kafka help you?

Kafka is event streaming platform, what does that mean?

Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:

1. To **publish** (write) and **subscribe to** (read) streams of events, including continuous import/export of your data from other systems.
2. To **store** streams of events durably and reliably for as long as you want.
3. To **process** streams of events as they occur or retrospectively.

Publish + Subscribe

At its heart lies the humble, immutable commit log, and from there you can subscribe to it, and publish data to any number of systems or real-time applications. Unlike messaging queues, Kafka is a highly scalable, fault tolerant distributed system, allowing it to be deployed for applications like managing passenger and driver matching at Uber, providing real-

time analytics and predictive maintenance for British Gas' smart home, and performing numerous real-time services across all of LinkedIn. This unique performance makes it perfect to scale from one app to company-wide use.

Store

An abstraction of a distributed commit log commonly found in distributed databases, Apache Kafka provides durable storage. Kafka can act as a 'source of truth', being able to distribute data across multiple nodes for a highly available deployment within a single data center or across multiple availability zones.

Process

An event streaming platform would not be complete without the ability to manipulate that data as it arrives. The Streams API within Apache Kafka is a powerful, lightweight library that allows for on-the-fly processing, letting you aggregate, create windowing parameters, perform joins of data within a stream, and more. Perhaps best of all, it is built as a Java application on top of Kafka, keeping your workflow intact with no extra clusters to maintain

P9

How does Kafka work in a nutshell?

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

P10

How The Kafka Project Handles Clients

Starting with the 0.8 release we are maintaining all but the jvm client external to the main code base. The reason for this is that it allows a small group of implementers who know the language of that client to quickly iterate on their code base on their own release cycle. Having these maintained centrally was becoming a bottleneck as the main committers can't hope to know every possible programming language to be able to perform meaningful code review and testing. This lead to a scenario where the committers were attempting to review and test code they didn't understand.

We are instead moving to the redis/memcached model which seems to work better at supporting a rich ecosystem of high quality clients.

We haven't tried all these clients and can't vouch for them. The normal rules of open source apply.

If you are aware of other clients not listed here (or are the author of such a client), please add it here. Please also feel free to help fill out information on the features the client supports, level of activity of the project, level of documentation, etc.

P12

Main Concepts and Terminology

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be "payments". Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the

topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

To make your data fault-tolerant and highly-available, every topic can be **replicated**, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.

Kafka Streams is a client library for processing and analyzing data stored in Kafka. It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management and real-time querying of application state.

Kafka Streams has a **low barrier to entry**: You can quickly write and run a small-scale proof-of-concept on a single machine; and you only need to run additional instances of your application on multiple machines to scale up to high-volume production workloads. Kafka Streams transparently handles the load balancing of multiple instances of the same application by leveraging Kafka's parallelism model.

Some highlights of Kafka Streams:

- Designed as a **simple and lightweight client library**, which can be easily embedded in any Java application and integrated with any existing packaging, deployment and operational tools that users have for their streaming applications.
- Has **no external dependencies on systems other than Apache Kafka itself** as the internal messaging layer; notably, it uses Kafka's partitioning model to horizontally scale processing while maintaining strong ordering guarantees.
- Supports **fault-tolerant local state**, which enables very fast and efficient stateful operations like windowed joins and aggregations.
- Supports **exactly-once** processing semantics to guarantee that each record will be processed once and only once even when there is a failure on either Streams clients or Kafka brokers in the middle of processing.
- Employs **one-record-at-a-time processing** to achieve millisecond processing latency, and supports **event-time based windowing operations** with late arrival of records.
- Offers necessary stream processing primitives, along with a **high-level Streams DSL** and a **low-level Processor API**.

We first summarize the key concepts of Kafka Streams.

Introduction

This article gives a glimpse of what exactly happens when a message is produced to Kafka, followed by how it is stored in Kafka and finally how it is consumed by a consumer.

Before that, let's go through some basic constructs and terminologies used in Kafka.

Apache Kafka is a distributed pub/sub messaging system where producers can publish messages to Kafka and consumers can subscribe to certain classes of messages and consume them. It is often regarded as a distributed commit log since messages published to Kafka are stored reliably in order until the retention period.

Message: Message is a record of information. Each message has an optional key which is used for routing the message to appropriate partition in a topic, a mandatory value which is the actual information. Both key and value of the message are arrays of bytes.

Producer: Producer is an application that is responsible for publishing messages to Kafka.

Broker: Broker is an application that is responsible for persisting messages. Consumers interact with the broker to consume messages published to topics.

Cluster: Brokers operate as part of the cluster to share the load and provide fault tolerance.

Topic: Topic is used for classification of messages into logical groups.

Partition: Each topic is divided into multiple partitions. Each partition is replicated across a configurable number of brokers to handle redundancy and scalability. Each partition has one leader which is responsible for reads and writes and zero or more followers which replicate the leader.

Offset: Each message is uniquely identified by a topic, partition it belongs and the offset number. Offset is a continually increasing integer that identifies a message uniquely given a topic and a partition. Messages are ordered within a partition by offset number.

Consumer: Consumer is an application that is responsible for consuming messages published on specific topics. Consumers operate as part of the consumer group sharing the load and providing fault tolerance.

Zookeeper: Zookeeper is a distributed coordination service that is used by Kafka to store broker metadata, topic metadata, etc...

P13

Stream Processing Topology

- A **stream** is the most important abstraction provided by Kafka Streams: it represents an unbounded, continuously updating data set. A stream is an ordered, replayable, and fault-tolerant sequence of immutable data records, where a **data record** is defined as a key-value pair.
- A **stream processing application** is any program that makes use of the Kafka Streams library. It defines its computational logic through one or more **processor topologies**, where a processor topology is a graph of stream processors (nodes) that are connected by streams (edges).
- A **stream processor** is a node in the processor topology; it represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in the topology, applying its operation to it, and may subsequently produce one or more output records to its downstream processors.

There are two special processors in the topology:

- **Source Processor**: A source processor is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors.

- **Sink Processor**: A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.

Note that in normal processor nodes other remote systems can also be accessed while processing the current record. Therefore the processed results can either be streamed back into Kafka or written to an external system.

Kafka Streams offers two ways to define the stream processing topology: the **Kafka Streams DSL** provides the most common data transformation operations such as `map`, `filter`, `join` and `aggregations` out of the box; the lower-level **Processor API** allows developers define and connect custom processors as well as to interact with state stores.

A processor topology is merely a logical abstraction for your stream processing code. At runtime, the logical topology is instantiated and replicated inside the application for parallel processing (see **Stream Partitions and Tasks** for details).

P14

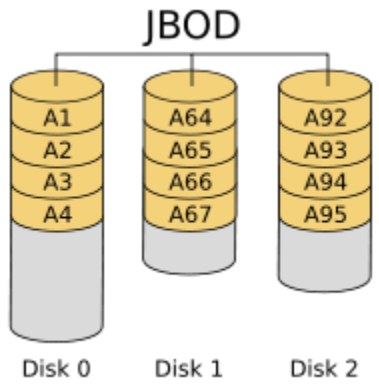
Persistence

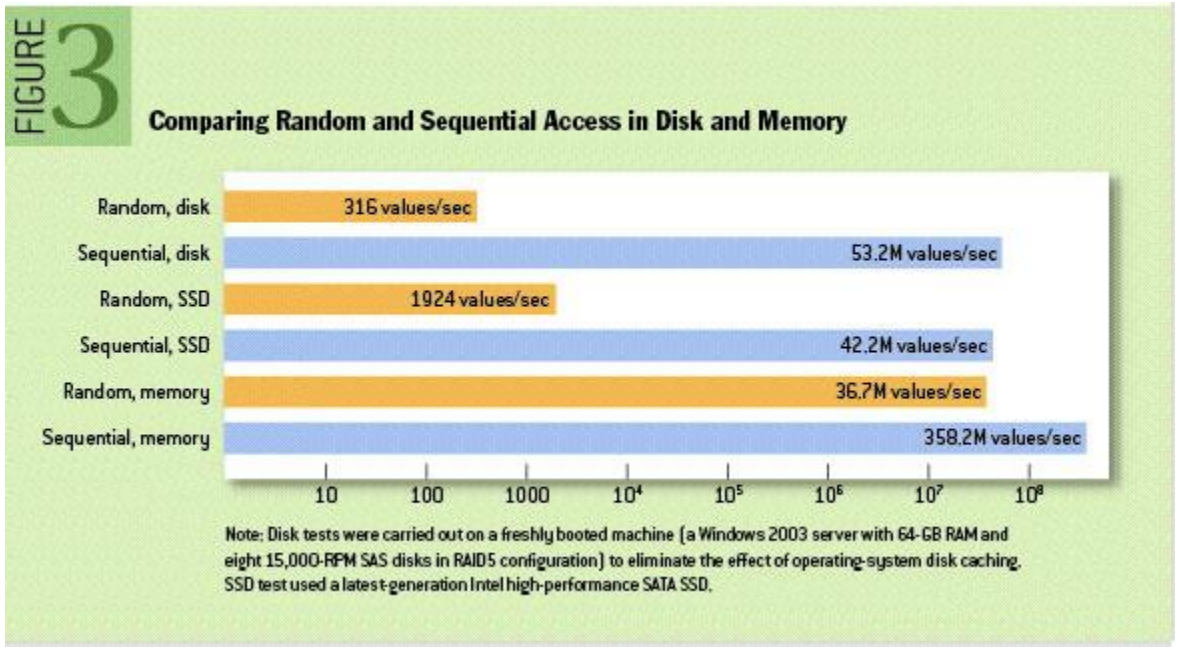
Don't fear the filesystem!

Kafka relies heavily on the file system for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact disks are both much slower and much faster than people expect depending on how they are used; and a properly designed disk structure can often be as fast as the network.

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a JBOD (derived from "**just a bunch of disks**") – [Non-RAID drive architectures] configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X. These linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further

discussion of this issue can be found in this ACM Queue article; they actually find that sequential disk access can in some cases be faster than random memory access!





P16

Message metadata

Tip #5: Use the power of record headers

Apache Kafka 0.11 introduced the concept of record headers. Record headers give you the ability to add some metadata about the Kafka record, without adding any extra information to the key/value pair of the record itself. Consider if you wanted to embed some information in a message, such as an identifier for the system from which the data originated. Perhaps you want this for lineage and audit purposes and in order to facilitate routing of the data downstream.

Why not just append this information to the key? Then you could extract the part needed and you would be able to route data accordingly. But adding artificial data to the key poses two potential problems.

1. First, if you are using a compacted topic, adding information to the key would make the record incorrectly appear as unique. Thus, compaction would not function as intended.
2. For the second issue, consider the effect if one particular system identifier dominates in the records sent. You now have a situation where you could have significant key skew. Depending on how you are consuming from the partitions, the uneven distribution of keys could have an impact on processing by increasing latency.

These are two situations where you might want to use headers. The original KIP proposing headers provides some additional cases as well:

- Automated routing of messages based on header information between clusters
- Enterprise APM tools (e.g., Appdynamics or Dynatrace) need to stitch in “magic” transaction IDs for them to provide end-to-end transaction flow monitoring.
- Audit metadata is recorded with the message, for example, the client-id that produced the record.
- Business payload needs to be encrypted end to end and signed without tamper, but ecosystem components need access to metadata to achieve tasks

P17

Today, in this Kafka SerDe article, we will learn the concept to create a custom serializer and deserializer with Kafka. Moreover, we will look at how serialization works in Kafka and why serialization is required. Along with this, we will see Kafka serializer example and Kafka deserializer example. In addition, this Kafka Serialization and Deserialization tutorial provide us with the knowledge of Kafka string serializer and Kafka object serializer.
Basically, **Apache Kafka** offers the ability that we can easily publish as well as subscribe to streams of records. Hence, we have the flexibility to create our own custom serializer as well as deserializer which helps

to transmit different data type using it.
So, let’s start Kafka Serialization and Deserialization

Apache Kafka SerDe

However, the process of converting an object into a stream of bytes for the purpose of transmission is what we call Serialization. Although, Apache Kafka stores as well as transmit these bytes of arrays in its queue.

Read Apache Kafka Use cases | Kafka Applications

Whereas, the opposite of Serialization is Deserialization. Here we convert bytes of arrays into the data type we desire. However, make sure Kafka offers serializers and deserializers for only a few data types, such as

- String
- Long
- Double
- Integer
- Bytes

Why Use Custom Serializer and Deserializer with Kafka?
Basically, in order to prepare the message for transmission from the producer to the broker, we use serializers. In other words, before transmitting the entire message to the broker, let the producer know how to convert the message into byte array we use serializers. Similarly, to convert the byte array back to the object we use the deserializers by the consumer.

P18

4. Implementation of Kafka SerDe

It is very important to implement the org.apache.kafka.common.serialization.Serializer interface to create a serializer class. Ans, for deserializer class, it is important to implement the org.apache.kafka.common.serialization.Deserializer

interface.

Let’s discuss Apache Kafka Architecture and its fundamental concepts

There are 3 methods for both Kafka serialization and deserialization interfaces:

Configure

At startup with configuration, we call Configure method.

b. Serialize/deserialize

For the purpose of Kafka serialization and deserialization, we use this method.

c. Close

While Kafka session is to be closed, we use Close method.

Read How to Create Kafka Clients

Serializer Interface With Kafka

```
public interface Serializer extends Closeable {

 void configure(Map<String, ?> var1, boolean var2);

 byte[] serialize(String var1, T var2);

 void close();

}
```

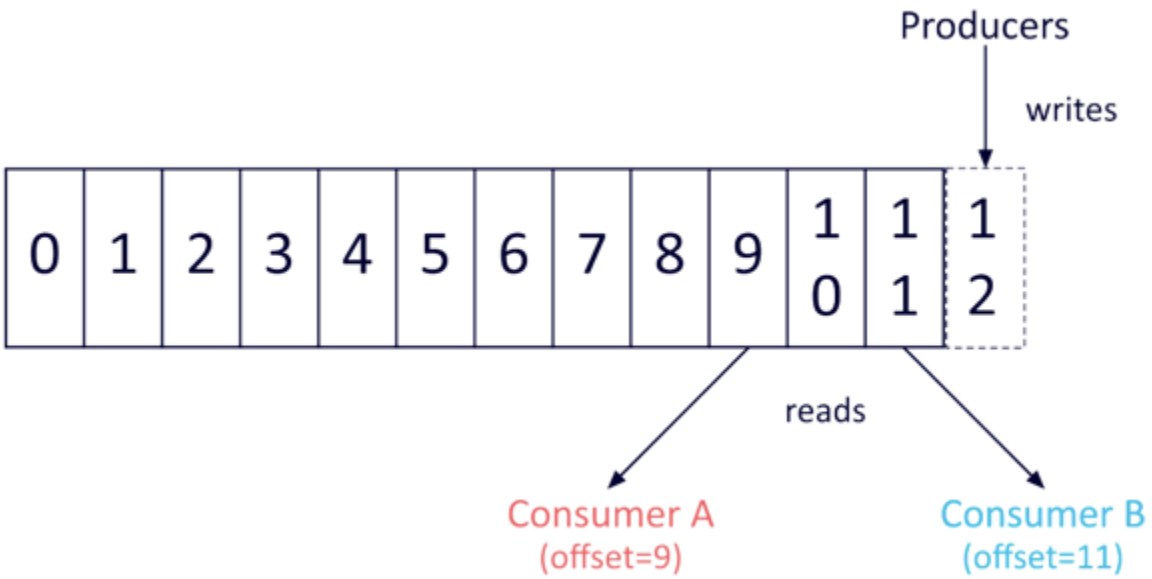
6. Deserializer Interface With Kafka

```
public interface Deserializer extends Closeable {

 void configure(Map<String, ?> var1, boolean var2);

 T deserialize(String var1, byte[] var2);

 void close();

}
```

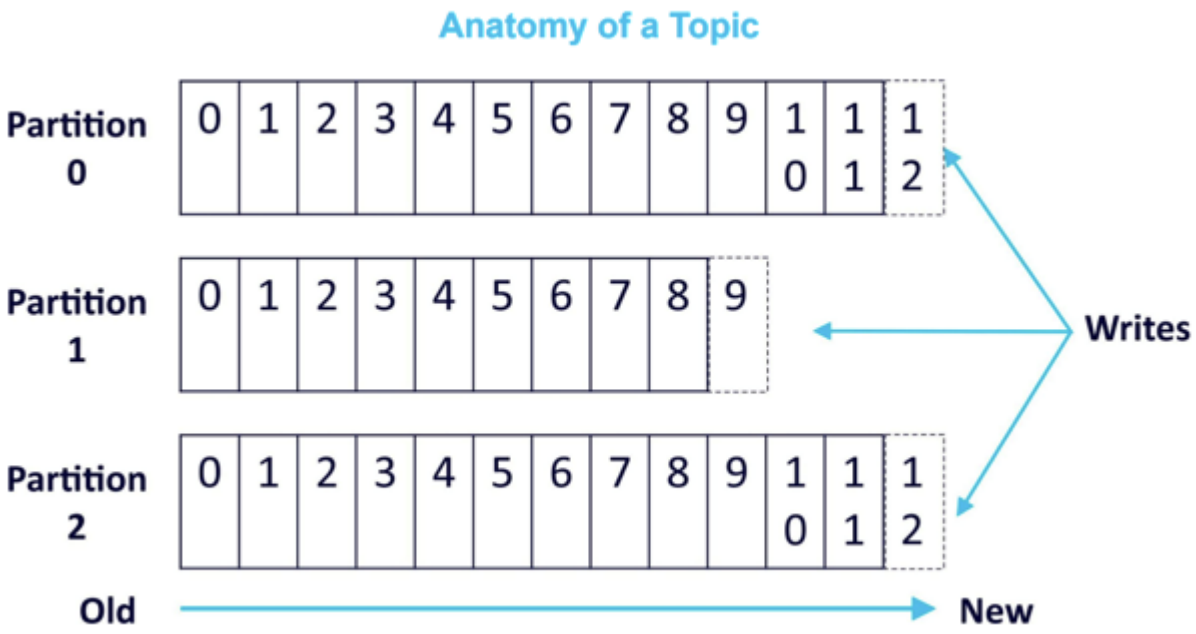
P26-27

While Apache Kafka has earned its reputation as a highly capable distributed streaming platform, it also features a bit of complexity when it comes to ensuring that data is stored – and can be retrieved – in the order that you want it to be.

To capture streaming data, Kafka publishes records to a *topic*, a category or feed name that multiple Kafka *consumers* can subscribe to and retrieve data. The Kafka cluster maintains a partitioned log for each topic, with all messages from the same *producer* sent to the same *partition* and added in the order they arrive. In this way, partitions are structured commit logs, holding ordered and immutable sequences of records. Each record added to a partition is assigned an *offset*, a unique sequential ID.



The challenge of receiving data in the order you prefer it in Kafka has a relatively simple solution: a partition maintains a strict order, and will always send data to consumers in the order it was added to the partition. However, Kafka does not maintain a total order of records across topics with multiple partitions.



For applications requiring total control of records, the solution is using a topic with just a single partition (this will also limit utilization to a single consumer process per consumer group).

Keep in mind that most applications will not require this level of control, and are better served by the technique of using per-partition ordering alongside data partitioning by key.

Examples of How to Use Kafka Partitioning and Keying

Let’s look at an example of what happens when we send data to multiple partitions. First, we’ll create a topic with 10 partitions, named “my-topic”:

```
./kafka-topics.sh --create \
 --zookeeper localhost:2181/kafka \
 --replication-factor 1 --partitions 10 \
 --topic my-topic
```

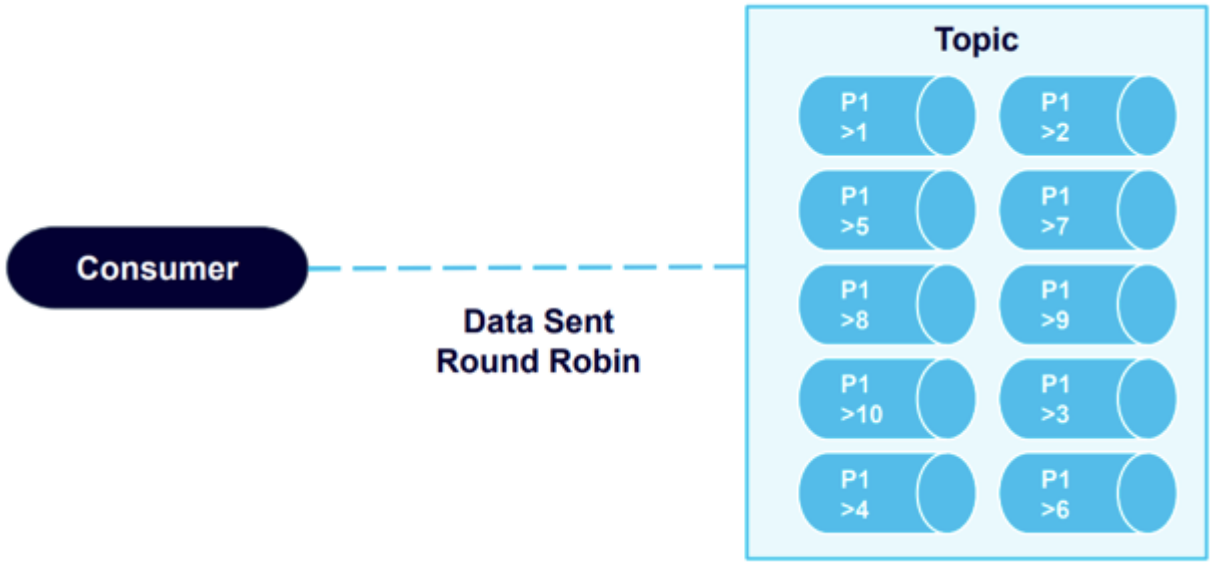
Next, we'll make a producer and send records containing the numbers "1" through "10".

```
./kafka-console-producer.sh \
 --broker-list localhost:9092 \
 --topic my-topic

> 1
> 2
> 3
> 4
> 5
> 6
> 7
> 8
> 9
> 10
```

These records are sent to the topic in order, but to ten different partitions. When we use the consumer to read this data from the beginning of the topic, it arrives quite out of order:

```
./kafka-console-consumer.sh \
 --bootstrap-server localhost:9092 \
 --topic my-topic \
 --from-beginning

> 2
> 5
> 9
> 3
> 4
> 7
> 6
> 10
> 8
> 1
```

This demonstrates the fact that data order is not guaranteed at the topic level – data is retrieved from all partitions in a round robin fashion.



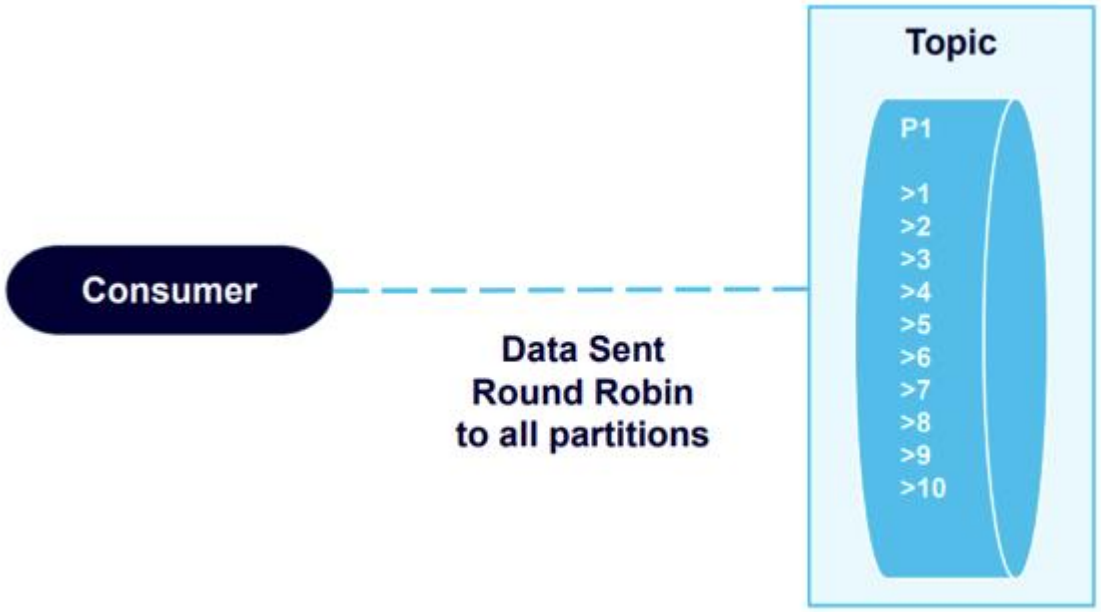
Now let's create a new example, using a topic with just one partition:

```
./kafka-topics.sh --create \
 --zookeeper localhost:2181/kafka \
 --replication-factor 1 --partitions 1 \
 --topic my-topic
```

We'll send the same "1" through "10" data:

```
./kafka-console-producer.sh \
 --broker-list localhost:9092 \
 --topic my-topic

> 1
> 2
> 3
> 4
> 5
> 6
> 7
> 8
> 9
> 10
```

This time, we retrieve the data in the order we sent it.



This validates Kafka’s guarantee of data order on the partition level. The producer sends messages in a specific order, the broker writes to the partition in that order, and consumers read the data in that order.

P28

When you are about to set up and run Kafka, the most important question to ask is what you want to achieve? In many businesses, the goal is to create a high performance and reliable setup in which the messages flowing within are both fast and in the correct order. That's why we sometimes receive the question regarding the number of partitions and the possibility to keep a strict ordering of the messages, even when you are using more than one?

The question is accurate since most businesses require things to be in the correct order, for example, businesses keeping track of user actions, metrics and user checkout flow etc.

The simple answer to this particular question is: Yes. If you want a strict order of **all** messages going to one topic, then you must use only **one** partition. However, a higher number of partitions is preferable for high throughput in Kafka, and these two factors can seem incompatible with each other.

But , in most cases, we have seen that people rather than the above statement, would prefer messages to be ordered based on a certain property in the message. This means that you, can use multiple partitions and still reach the result you wish for, i.e a strict order of messages while using numerous partitions.

For example:

If you have a topic with *user actions* which must be ordered, but only ordered per user, the need for using only one partition is no longer needed. This action can be supported by having multiple partitions but using a consistent message key, for example, user id. This will guarantee that all messages for a certain user always ends up in the same partition and thus is ordered.

A consumer will consume from one or more partition, but you will never have two consumers consuming from one partition. So in the case above, if you have 10 partitions and you use 10 consumers, one consumer will get all messages related to the same user and thus be processed in order.

So the answer is as simple as this:

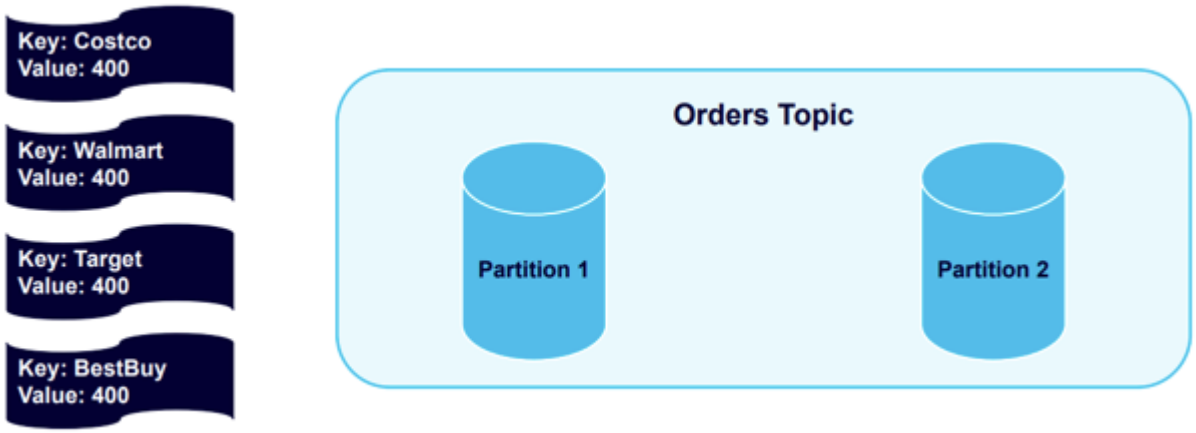
If all messages must be ordered within one topic, use one partition, but if messages can be ordered per a certain property, set a consistent message key and use multiple partitions.

This way you can keep your messages in strict order and keep high Kafka throughput.

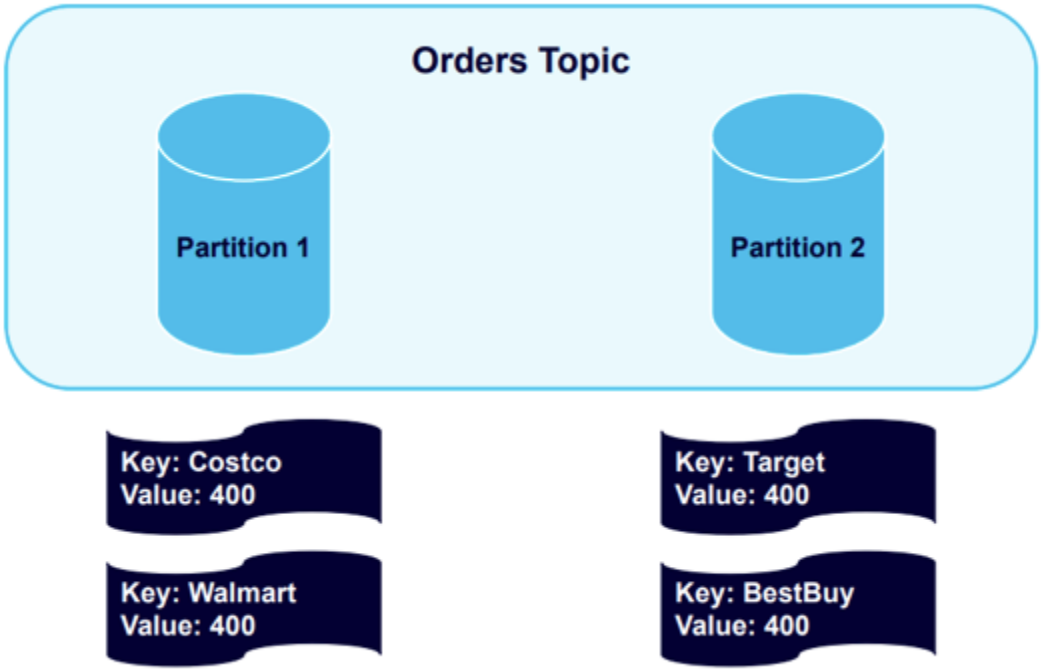
And as always, feel free to send us any questions or feedback you might have

P29-37

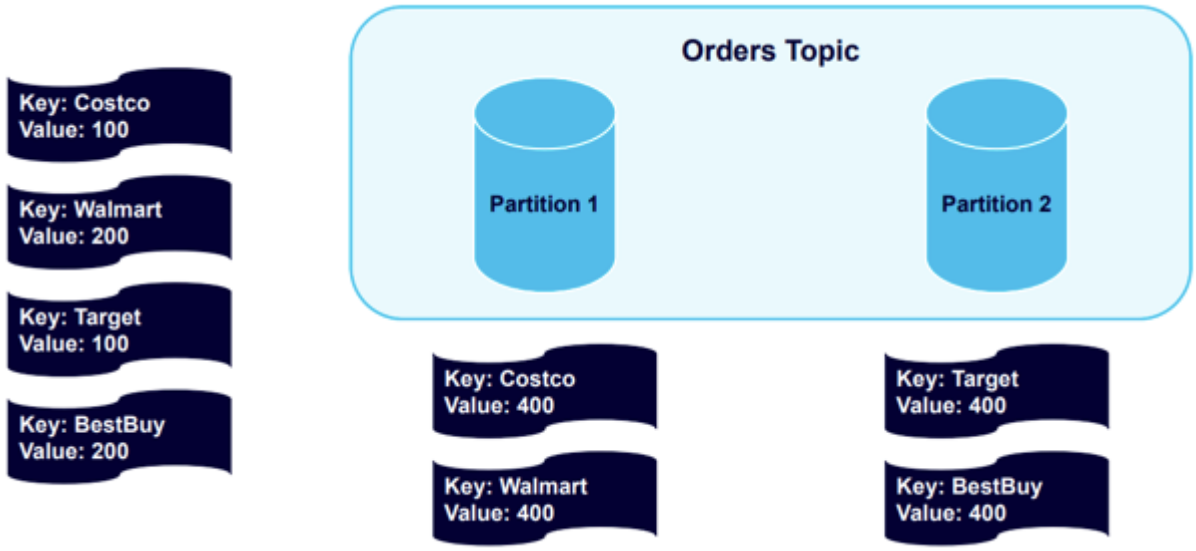
Next, let’s use an example that introduces keying, which allows you to add keys to producer records. We’ll send four messages that include keys to a Kafka topic with two partitions:



The four different keys – Costco, Walmart, Target, and Best Buy, and hashed and distributed across the cluster into the partitions:



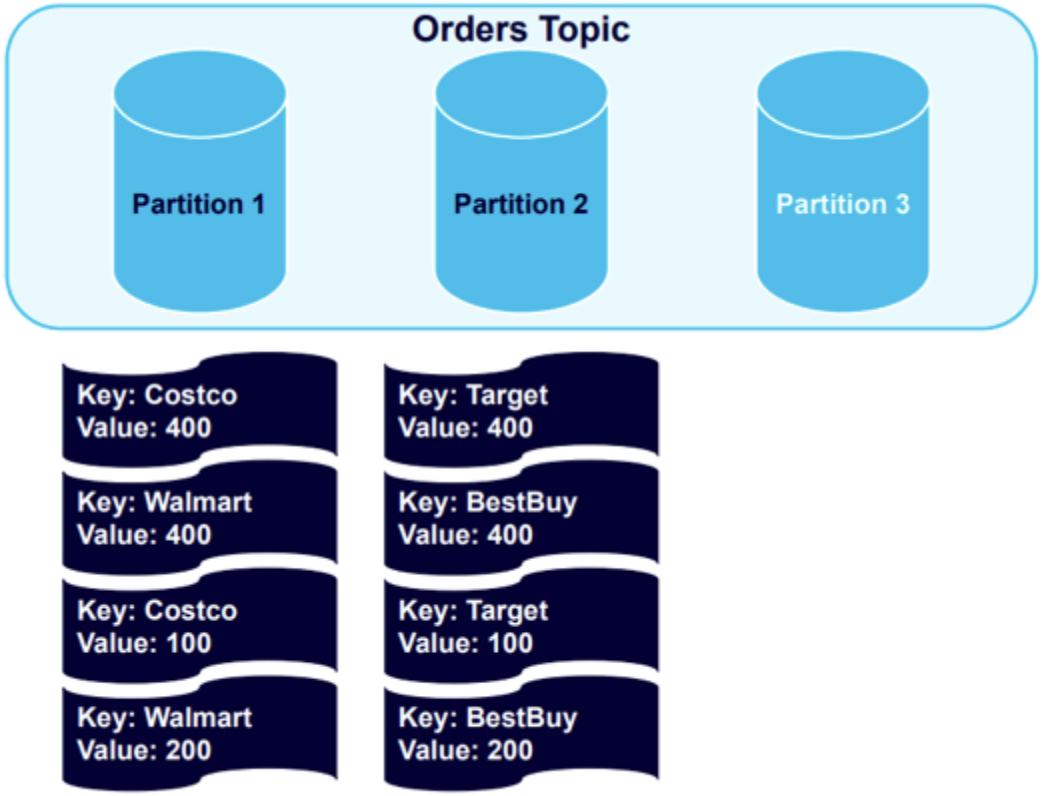
Now we'll send another four messages:



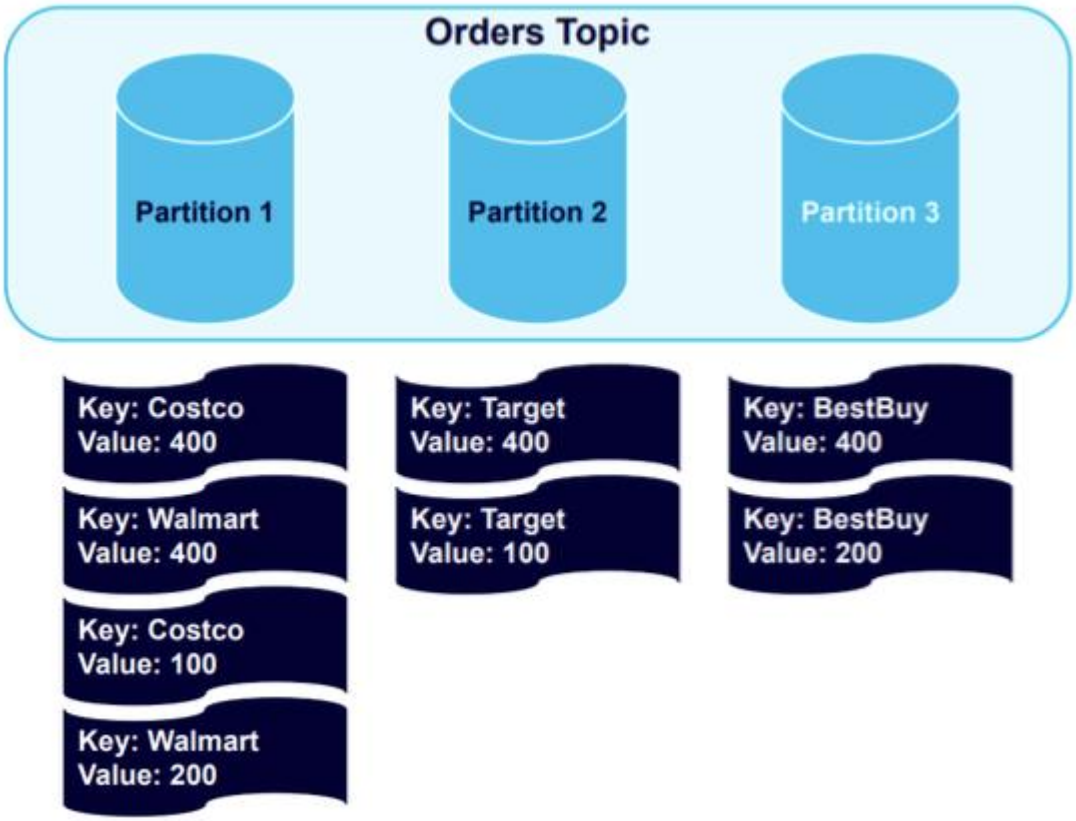
Kafka will send messages to the partitions that already use the existing key:

All Costco or Walmart records go in Partition 1, and all Target or Best Buy records go in Partition 2. The records are in the order that they were sent to those partitions.

Next, let's see what happens if we add more partitions to the cluster, which we might want to do in order to balance the data in a healthier way.



With the new partition added, we'll trigger a rebalance event:



The data remains nicely structured by key, with all Best Buy data balanced to Partition 3. It will remain so if we add four more messages to the topic:

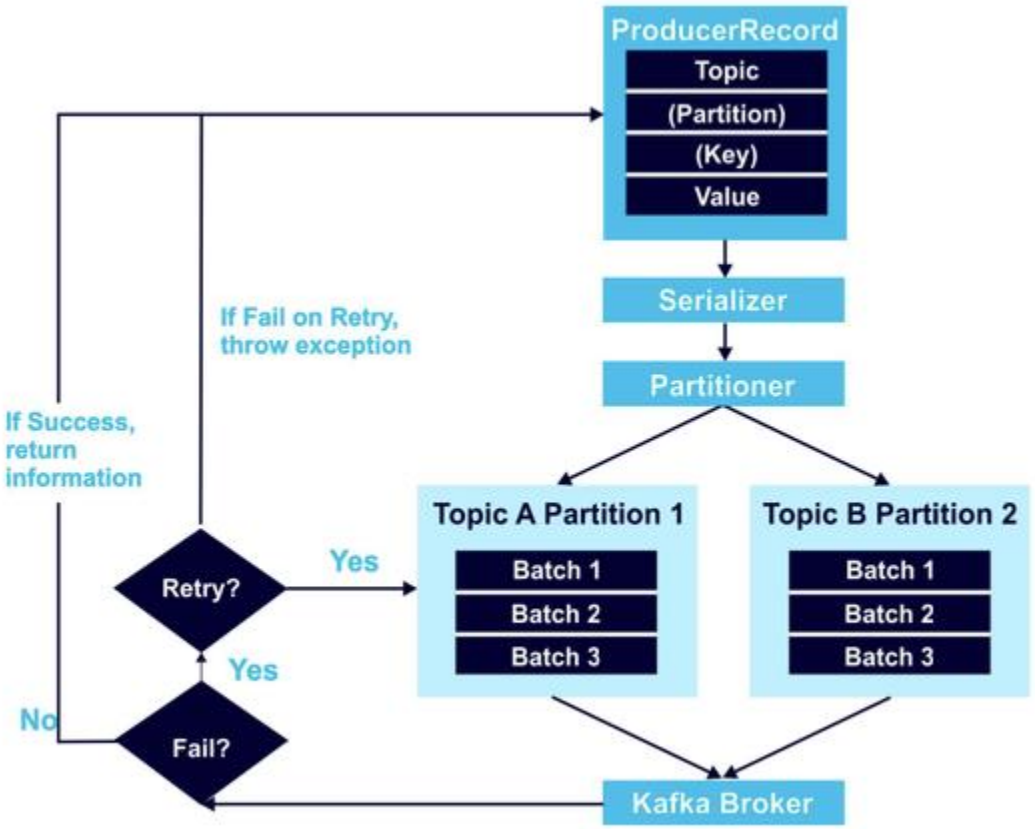
Data goes to the partitions holding the established keys. However, because one partition is holding twice as much data as the others, it makes logical sense to add a fourth partition and trigger another rebalance:

The result is a healthy balancing of the data sets.

Ensuring that Data is Always Sent in Order

There are a number of additional issues that can cause data to arrive out of order in Kafka, including broker or client failures and disorder produced by reattempts to send data. To address these , let's first take a closer look at the Kafka producer.

Here is a high-level overview showing how a Kafka producer works:



At minimum, a ProducerRecord object includes the topic to send the data to, and a value. It can also include a specified partition to use, and key. At stated in the examples above, I recommend that you always use keys. If you don't, data will be distributed round robin to any partition with no organization in place. Data from the ProducerRecord is next encoded with the Serializer, and then the Partitioner algorithm decides where the data goes.

The retry mechanism, outlined on the left-hand side of the above diagram, is an area where data order issues can frequently occur. For example, say you attempt to send two records to Kafka, but one fails due to a network issue and the

other goes through. When you try resending the data, there's a risk that data will be out of order, because you're now sending two requests to Kafka simultaneously.

You can resolve this issue by setting max.in.flight.requests.per.connection to 1. If this is set to more than one (and the retries parameter is nonzero), the broker could fail to write the first message batch, successfully write the second because it was allowed to also be in-flight, and then successfully retry the first batch, swapping their order to one you didn't intend. In contrast, setting max.in.flight.requests.per.connection to 1 ensures that those requests occur and one after the other and in order.

In scenarios where order is crucial, I recommend setting in.flight.requests.per.session to 1; this ensures that additional messages won't be sent while a message batch is retrying. However, this tactic severely limits producer throughput, and should *only* be used if order is essential. Setting allowed retries to zero may seem like a possible alternative, however, if the impact on system reliability makes it a non-option.

P38

For data durability, the `KafkaProducer` has the configuration setting `acks`. The `acks` configuration specifies how many acknowledgments the producer receives to consider a record delivered to the broker. The options to choose from are:

- `none`: The producer considers the records successfully delivered once it sends the records to the broker. This is basically "fire and forget."
- `one`: The producer waits for the lead broker to acknowledge that it has written the record to its log.
- `all`: The producer waits for an acknowledgment from the lead broker and from the following brokers that they have successfully written the record to their logs.

As you can see, there is a trade-off to make here—and that's by design because different applications have different requirements. You can opt for higher throughput with a chance for data loss, or you may prefer a very high data durability guarantee at the expense of a lower throughput.

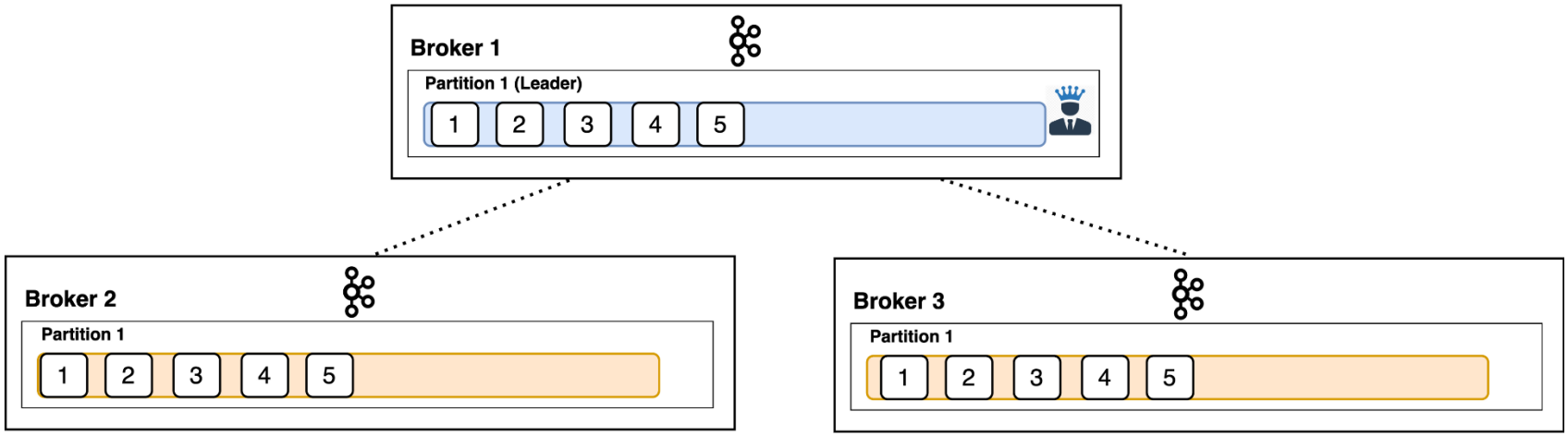
P39

Replication

To best understand these configs, it's useful to remind ourselves of Kafka's replication protocol.

For each partition, there exists one leader broker and *n* follower brokers.
The config which controls how many such brokers *(1 + N)* exist is `replication.factor`. That's the total amount of times the data inside a single partition is replicated across the cluster.

The default and typical recommendation is three.



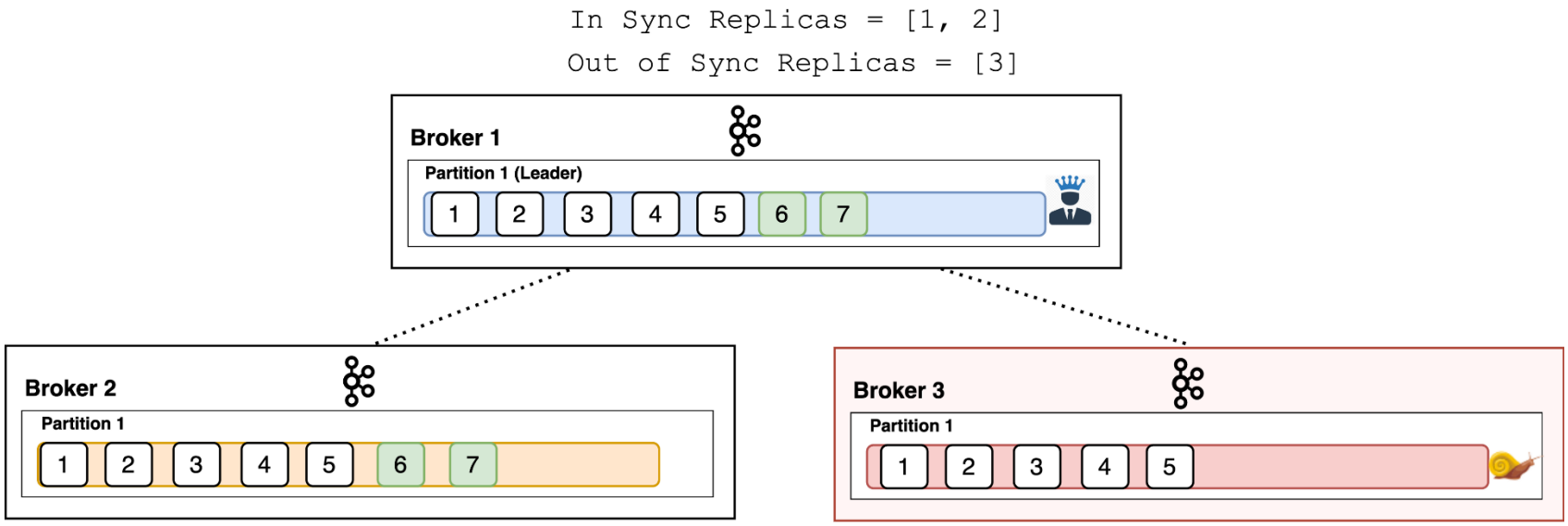
Producer clients only write to the leader broker — the followers asynchronously replicate the data. Now, because of the messy world of distributed systems, we need a way to tell whether these followers are managing to keep up with the leader — do they have the latest data written to the leader?

P40

In-sync replicas

An *in-sync replica* (ISR) is a broker that has the latest data for a given partition. A *leader* is always an in-sync replica. A *follower* is an in-sync replica only if it has fully caught up to the partition it's following. In other words, it can't be behind on the latest records for a given partition.

If a follower broker falls behind the latest data for a partition, we no longer count it as an in-sync replica.



Note that the way we determine whether a replica is in-sync or not is a bit more nuanced — it's not as simple as "Does the broker have the latest record?" Discussing that is outside the scope of this article. For now, trust me that red brokers with snails on them are out of sync.

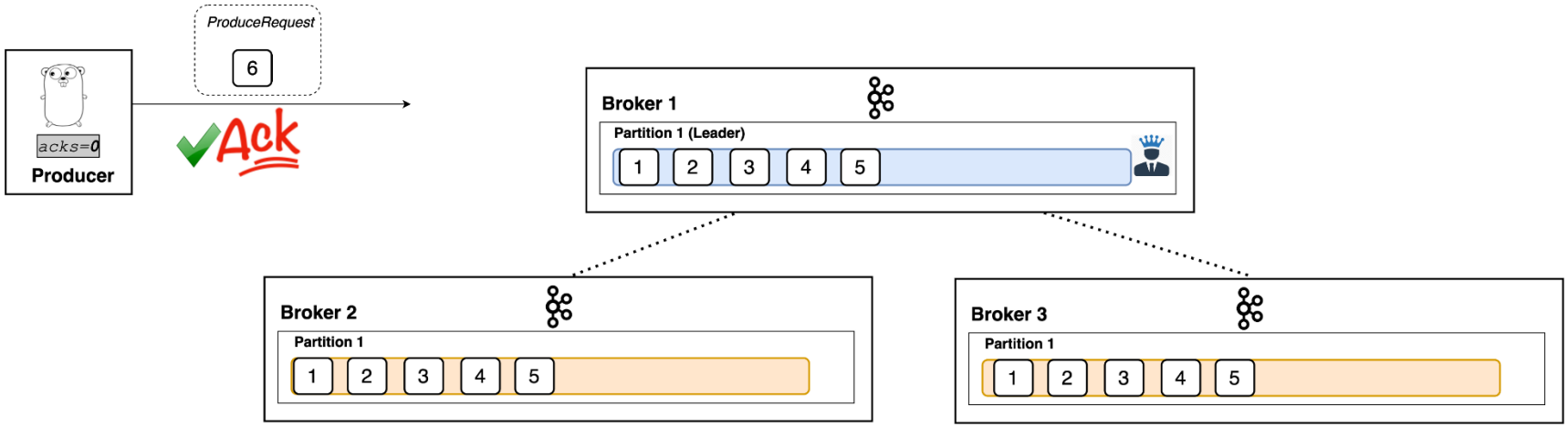
P41

Acknowledgements

The `acks` setting is a client (producer) configuration. It denotes the number of brokers that must receive the record before we consider the write as successful. It support three values — `0`, `1`, and `all`.

‘acks=0’

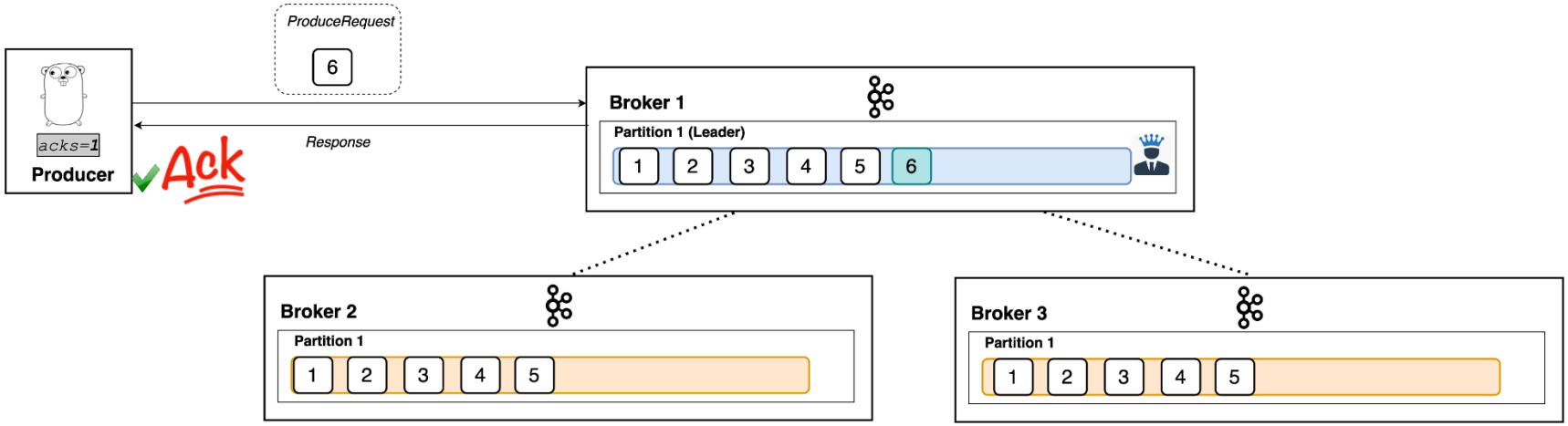
With a value of `0`, the producer won’t even wait for a response from the broker. It immediately considers the write successful the moment the record is sent out



P42

‘acks=1’

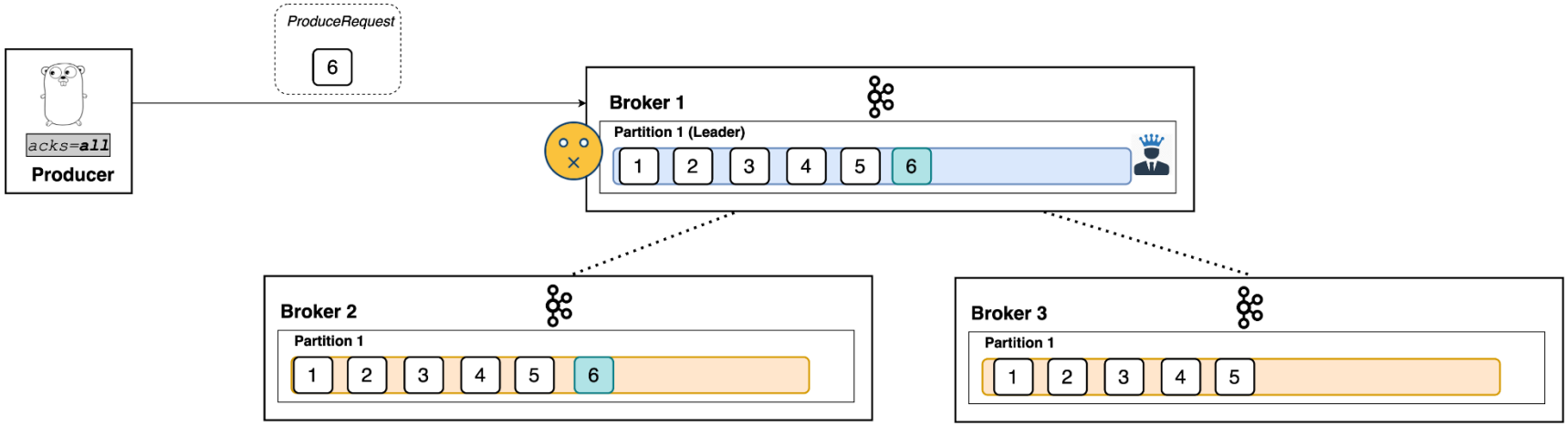
With a setting of `1`, the producer will consider the write successful when the leader receives the record. The leader broker will know to immediately respond the moment it receives the record and not wait any longer



P43

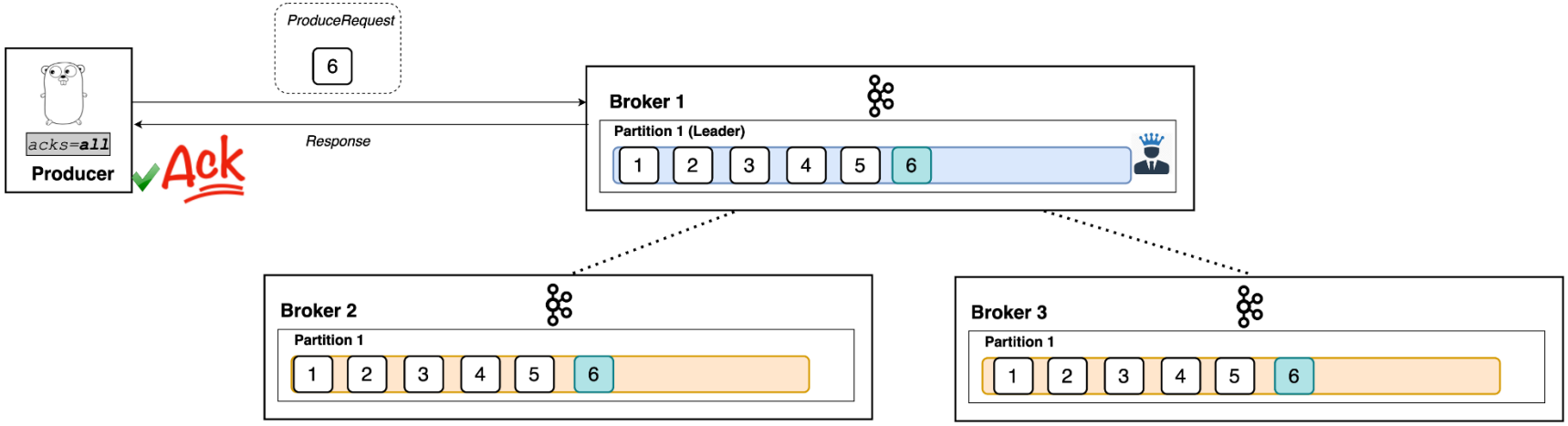
‘acks=all’

When set to `all`, the producer will consider the write successful when all of the in-sync replicas receive the record. This is achieved by the leader broker being smart as to when it responds to the request — it’ll send back a response once all the in-sync replicas receive the record themselves



P44

Like I said, the leader broker knows when to respond to a producer that uses `acks=all`



Acks's utility

As you can tell, the `acks` setting is a good way to configure your preferred trade-off between durability guarantees and performance.

If you'd like to be sure your records are nice and safe — configure your acks to `all`.

If you value latency and throughput over sleeping well at night, set a low threshold of `0`. You may have a greater chance of losing messages, but you inherently have better latency and throughput.

P45

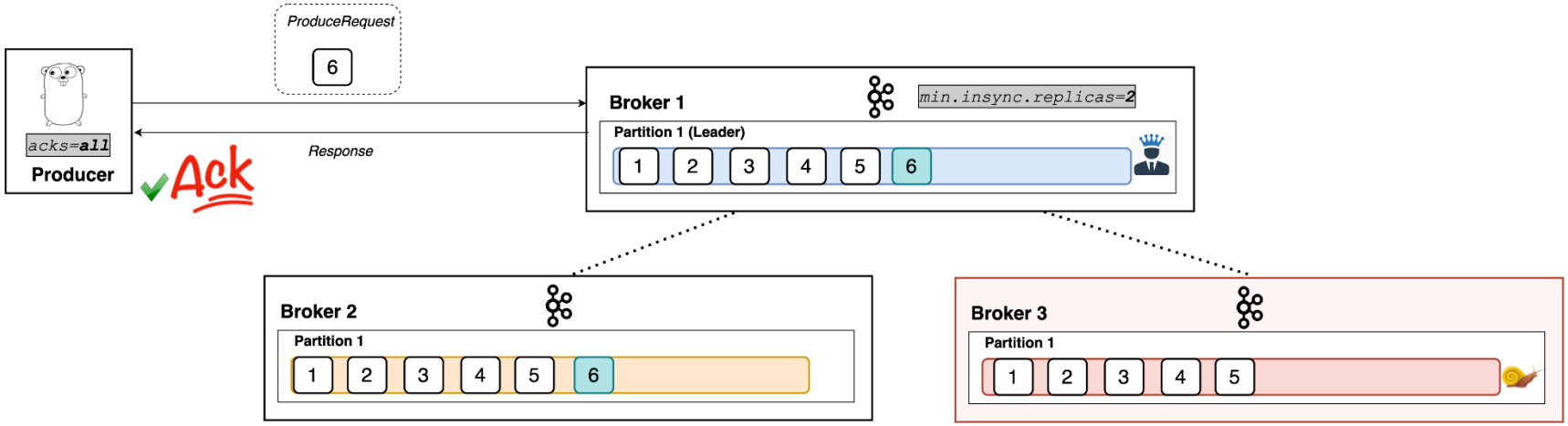
Minimum In-Sync Replica

There's one thing missing with the `acks=all` configuration in isolation.
If the leader responds when all the in-sync replicas have received the write, what happens when the leader is the only in-sync replica? Wouldn't that be equivalent to setting `acks=1` ?

This is where `min.insync.replicas` comes to shine!

`min.insync.replicas` is a config on the broker that denotes the minimum number of in-sync replicas required to exist for a broker to allow `acks=all` requests. That is, all requests with `acks=all` won't be processed and receive an error response if the number of in-sync

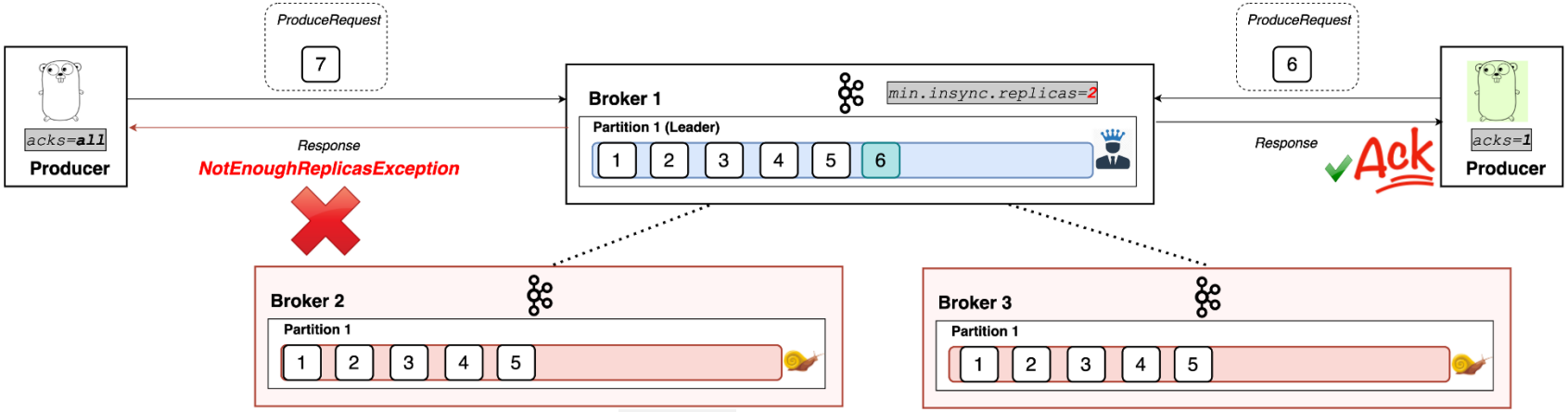
replicas is below the configured minimum amount. It acts as a sort of gatekeeper to ensure scenarios like the one described above can't happen



P46

As shown, `min.insync.replicas=X` allows `acks=all` requests to continue to work when at least *x* replicas of the partition are in sync. Here, we saw an example with two replicas.

But if we go below that value of in-sync replicas, the producer will start receiving exceptions



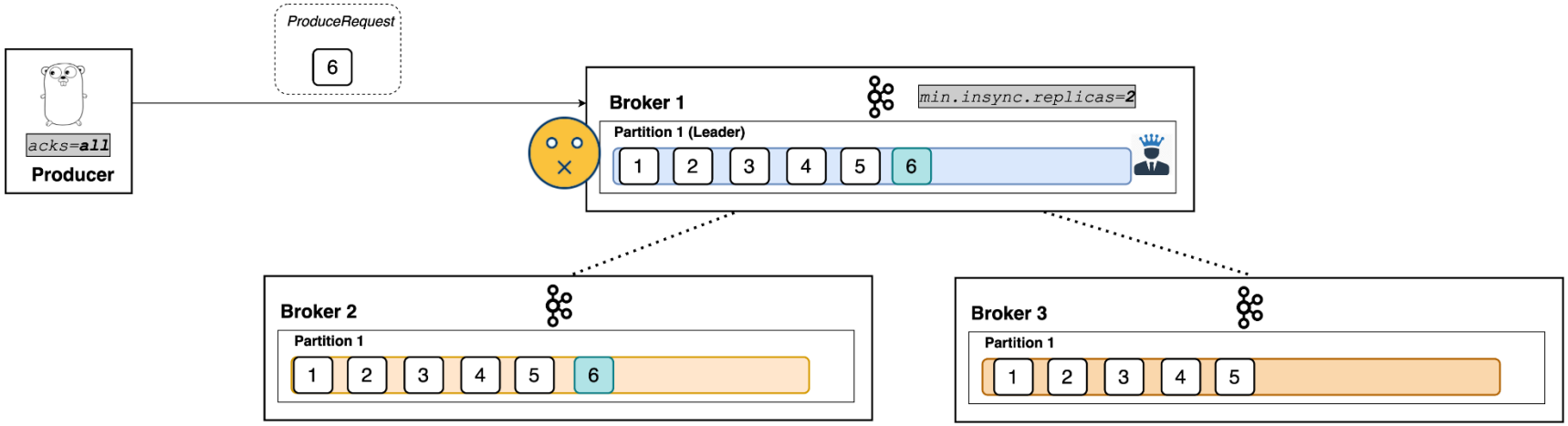
As you can see, producers with `acks=all` can't write to the partition successfully during such a situation. Note, however, that producers with `acks=0` or `acks=1` continue to work just fine.

P47

Caveat

A common misconception is that `min.insync.replicas` denotes how many replicas need to receive the record in order for the leader to respond to the producer. That's not true — the config is the *minimum* number of in-sync replicas required to exist in order for the request to be processed.

That is, if there are three in-sync replicas and `min.insync.replicas=2`, the leader will respond only when all three replicas have the record



P48

Each of the projects I've worked on in the last few years has involved a distributed message system such as AWS SQS, AWS Kinesis and more often than not Apache Kafka. In designing these systems, we invariably need to consider questions such as:

How do we guarantee all messages are processed?

How do we avoid or handle duplicate messages?

These simple questions are surprisingly hard to answer. To do so, we need to delve into how producers and consumers interact with distributed messaging systems. In this post I'll be looking at message processing guarantees and the implications these have when designing and building systems around distributed messages systems. I will be specifically concentrating on the Apache Kafka platform as it's such a popular choice and the one I am most familiar with.

To start with, let's look at the basic architecture of a distributed message system.

A producer process reads from some data source which may or may not be local, then writes to the messaging system over the network. The messdaging system persists the message, typically in multiple locations for redundancy. One or more consumers poll the messaging system over the network, receive batches of new messages and perform some action on these messages, often transforming the data and writing to some other remote data store, possibly back to the messaging system. This basic design applies to Apache Kafka, Apache Pulsar, AWS Kinesis, AWS SQS, Google Cloud Pub/Sub and Azure Event Hubs among others.

P49

If you read the documentation for any of these systems, you will fairly quickly come across the concept of message processing guarantees. These fall under the following categories:

- **No guarantee** — No explicit guarantee is provided, so consumers may process messages once, multiple times or never at all.
- **At most once** — This is “best effort” delivery semantics. Consumers will receive and process messages exactly once or not at all.
- **At least once** — Consumers will receive and process every message, but they may process the same message more than once.
- **Effectively once** — Also contentiously known as exactly once, this promises consumers will process every message once.

At this point, you may be asking why is this so complicated. Why isn’t it always effectively once? What’s causing messages to go missing or appear more than once? The short answer to this is system behavior in the face of failure. The key word in describing these architectures is distributed. Here is a small subset of failure scenarios that you will need to consider:

- Producer failure
- Consumer publish remote call failure
- Messaging system failure
- Consumer processing failure

Your consumer process could run out of memory and crash while writing to a downstream database; your broker could run out of disk space; a network partition may form between ZooKeeper instances; a timeout could occur publishing messages to Kafka. These types of failures are not just hypothetical — they can and will happen with any non-trivial system in any and all environments including production. How these failures are handled determines the processing guarantee of the system as a whole.

No guarantee

A system that provides no guarantee means any given message could be processed once, multiple times or not at all. With Kafka a simple scenario where you will end up with these semantics is if you have a consumer with enable.auto.commit set to true (this is the default) and for each batch of messages you *asynchronously* process and save the results to a database.

With auto commit enabled, the consumer will save offsets back to Kafka periodically at the start of subsequent poll calls. The frequency of these commits is determined by the configuration parameter auto.commit.interval.ms. If you save the messages to the database then the application crashes before the progress is saved, you will reprocess those messages again the next run and save them to the database twice. If progress is saved prior to the results being saved to the database, then the program crashes, these messages will not be reprocessed in the next run meaning you have data loss.

At most once

At most once guarantee means the message will be processed exactly once, or not at all. This guarantee is often known as “best effort” semantics.

1. A common example that results in at most once semantics is where a producer performs a ‘fire-and-forget’ approach sending a message to Kafka with no retries and ignoring any response from the broker. This approach is useful where progress is a higher priority than completeness.

2. A producer saves its progress reading from a source system first, then writes data into Kafka. If the producer crashes before the second step, the data will never be delivered to Kafka.

3. A consumer receives a batch of messages from Kafka, transforms these and writes the results to a database. The consumer application has enable.auto.commit set to false and is programmed to commit their offsets back to Kafka prior to writing to the database. If the consumer fails after saving the offsets back to Kafka but before writing the data to the database, it will skip these records next time it runs and data will be lost.

At least once

At least once guarantee means you will definitely receive and process every message, but you may process some messages additional times in the face of a failure. Here’s a few examples of some failure scenarios that can lead to at-least-once semantics:

1. An application sends a batch of messages to Kafka. The application never receives a response so sends the batch again. In this case it may have been the first batch was successfully saved, but the acknowledgement was lost, so the messages end up being added twice.

2. An application processes a large file containing events. It starts processing the file sending a message to Kafka for each event. Half way through processing the file the process dies and is restarted. It then starts processing the file again from the start and only marks it as processed when the whole file has been read. In this case the events from the first half of the file will be in Kafka twice.

3. A consumer receives a batch of messages from Kafka, transforms these and writes the results to a database. The consumer application has enable.auto.commit set to false and is programmed to commit their offsets back to Kafka once the database write succeeds. If the consumer fails after writing the data to the database but before saving the offsets back to Kafka, it will reprocess the same records next time it runs and save them to the database once more.

P51

Consumers read from any single partition, allowing you to scale throughput of message consumption in a similar fashion to message production. Consumers can also be organized into consumer groups for a given topic — each consumer within the group reads from a unique partition and the group as a whole consumes all messages from the entire topic. If you have more consumers than partitions then some consumers will be idle because they have no partitions to read from. If you have more partitions than consumers then consumers will receive messages from

multiple partitions. If you have equal numbers of consumers and partitions, each consumer reads messages in order from exactly one partition.

Before getting into the code, we should review some basic concepts. In Kafka, each topic is divided into a set of logs known as *partitions*. Producers write to the tail of these logs and consumers read the logs at their own pace. Kafka scales topic consumption by distributing partitions among a *consumer group*, which is a set of consumers sharing a common group identifier. The diagram below shows a single topic with three partitions and a consumer group with two members. Each partition in the topic is assigned to exactly one member in the group.

While the old consumer depended on Zookeeper for group management, the new consumer uses a group coordination protocol built into Kafka itself. For each group, one of the brokers is selected as the *group coordinator*. The coordinator is responsible for managing the state of the group. Its main job is to mediate partition assignment when new members arrive, old members depart, and when topic metadata changes. The act of reassigning partitions is known as *rebalancing* the group.

P52

When a group is first initialized, the consumers typically begin reading from either the earliest or latest offset in each partition. The messages in each partition log are then read sequentially. As the consumer makes progress, it *commits* the offsets of messages it has successfully processed. For example, in the figure below, the consumer’s position is at offset 6 and its last committed offset is at offset 1.

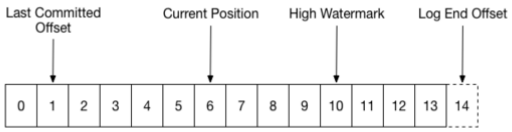


Figure 2: The Consumer's Position in the Log

When a partition gets reassigned to another consumer in the group, the initial position is set to the last committed offset. If the consumer in the example above suddenly crashed, then the group member taking over the partition would begin consumption from offset 1. In that case, it would have to reprocess the messages up to the crashed consumer's position of 6.

The diagram also shows two other significant positions in the log. The log end offset is the offset of the last message written to the log. The high watermark is the offset of the last message that was successfully copied to all of the log's replicas. From the perspective of the consumer, the main thing to know is that you can only read up to the high watermark. This prevents the consumer from reading un-replicated data which could later be lost.

P53

When the group coordinator—a broker that is selected to manage the state of a given consumer group—receives an `OffsetCommitRequest` by the application, it appends the request to the `__consumer_offsets` topic, which is then replicated to other Kafka brokers in the cluster based on the `offsets.topic.replication.factor` config parameter. This specifies the replication factor of the offset topic and affects the amount of brokers the offset commit will be replicated to.

The offset commit can be controlled from the broker configuration. You can set the `offsets.commit.required.acks`, which stands for the number of acknowledgments that are required before the offset commit can be accepted, as well as `offsets.commit.timeout.ms`, which defaults to five seconds.

P54

In the diagram above, you can see the details on a consumer group called `my-group`. The command output shows the details per partition within the topic. Two main columns that are worth mentioning are the `CURRENT-OFFSET`, which is the current max offset of the consumer on a given partition, and the `LOG-END-OFFSET`, which is the offset of the latest message in the partition.

P55

Lag is a key performance indicator (KPI) for Kafka. When building an event streaming platform, the consumer group lag is one of the crucial metrics to monitor.

As mentioned earlier, when an application consumes messages from Kafka, it commits its offset in order to keep its position in the partition. When a consumer gets stuck for any reason—for example, an error, rebalance, or even a complete stop—it can resume from the last committed offset and continue from the same point in time.

Therefore, lag is the delta between the last committed message to the last produced message. In other words, lag indicates how far behind your application is in processing up-to-date information.

To make matters worse, remember that Kafka persistence is based on retention, meaning that if your lag persists, you will lose data at some point in time. The goal is to keep lag to a minimum.

P56

Kafka Lag Exporter

Kafka Lag Exporter had native Kubernetes support and contained time-based lag monitoring out of the box. At the time, it was in beta and it didn't fit our predefined guidelines.

Product of light-bend Company – also has another famous product -> aka framework

Burrow

Burrow is an active LinkedIn project with more than 2,700 stars on GitHub. It has an active community, Gitter channel, great wiki page, and multiple extension projects. Burrow is battle tested and works in production at LinkedIn.

Remora

Remora was created after Zalando spent some time using Burrow. It has Datadog and CloudWatch integration, and it's a wrap around the Kafka consumer group command—which we already had.

P57

This dashboard enables us to visualize the lag by number of messages, the producer rate, the consumer rate, and a

comparison between the two

Another useful dashboard is the partition analysis dashboard.

With the previous solution, we couldn’t identify exactly which partitions were lagging behind, but with this dashboard, we can identify exactly which partition has the highest lag, correlate it with the partition leader, and potentially shed some light on the incident and find a misbehaving broker.

More importantly, we were able to achieve our end game: *time-based* metrics.

P58

The most important metrics to collect are the last consumer message and the last produced message.

P59

We know the lag per partition, per topic, and per consumer group. We also know the producer rate, which is our biggest assumption in the equation. We assume that the producer rate is stable—like a normal operating producer.

So if we will calculate the difference between the last consumed message and last produced message, and divide it by the producer rate, we will get the lag—in time units!

P60

In the diagram above is a Kafka timeline. The producer produces one message a minute. At 12:00 a.m., the produced offset was `134`. Ten minutes later at 12:10 a.m., the offset was `144`. Ten minutes after that, it was `154`, which is the current time.

While producing those messages, our consumer consumed messages at a given rate. Now, it just consumed the message that had the message offset of `134`, which we already know was produced 20 minutes ago. It can then be assumed that our consumer is lagging 20 minutes behind the producer, as it just read a message that we produced 20 minutes ago. This is how we managed to get time-based lag metrics.

P61

The graph provides you the overall produced message count and consumed message count across all the consumer groups for a topic within the selected time range. Any discrepancy in the produced and consumed message count is highlighted in red.

In the preceding image, the linear formation represents the number of messages produced in last one hour, and the filled area represents the number of messages consumed in last one hour with a granularity of 30 seconds. The blue area signifies that all the produced messages are consumed. The red area represents a discrepancy in the produced and consumed message count and can either mean that messages are over consumed or under consumed.

In the image, there are two red areas. The first red area, from the left, indicates that the number of consumed messages is more than the number of produced messages. This represents an overconsumption of messages which can occur

when a consumer group offset is reset to an older offset to reprocess messages, or when producers or consumers are shut down in an unclean manner.

The last red area indicates that the number of consumed messages is less than the number of produced messages. This represents an under consumption of messages which can occur when the consumer group offset is set to a newer offset causing the consumer group to skip processing some messages.

The far right section of the graph shows the current processing window where consumers are still consuming the produced messages. Therefore this area is expected to be marked red and indicates an under consumption of messages.

All other areas in the image are blue, which indicates that all the produced messages are consumed.

P63

Zookeeper Dependency

A running Apache ZooKeeper cluster is a key dependency for running Kafka

Apache Zookeeper is a distributed, open-source configuration, synchronization service along with naming registry for distributed applications.

But when using ZooKeeper alongside Kafka, there are some important best practices to keep in mind.

ZooKeeper stores a lot of shared information about **Kafka Consumers** and Kafka **Brokers**,

Kafka Brokers

Below given are the roles of ZooKeeper in **Kafka Broker**:

i. State
Zookeeper determines the state. That means, it notices, if the Kafka Broker is alive, always when it regularly sends heartbeats requests. Also, while the Broker is the constraint to handle replication, it must be able to follow replication needs.

ii. Quotas
In order to have different producing and consuming quotas, Kafka Broker allows some clients. This value is set in ZK under /config/clients path. Also, we can change it in bin/kafka-configs.sh script.

iii. Replicas
However, for each topic, Zookeeper in Kafka keeps a set of in-sync replicas (ISR). Moreover, if somehow previously

selected leader node fails then on the basis of currently live nodes Apache ZooKeeper will elect the new leader.
Have a look at Apache Kafka Career Scope with Salary trends

iv. Nodes and Topics Registry
Basically, Zookeeper in Kafka stores nodes and topic registries. It is possible to find there all available brokers in Kafka and, more precisely, which Kafka topics are held by each broker, under /brokers/ids and /brokers/topics zNodes, they're stored. In addition, when it's started, Kafka broker create the register automatically.

b. Kafka Consumers

i. Offsets
ZooKeeper is the default storage engine, for consumer offsets, in Kafka's 0.9.1 release. However, all information about how many messages Kafka consumer consumes by each consumer is stored in ZooKeeper.

ii. Registry
Consumers in Kafka also have their own registry as in the case of Kafka Brokers. However, same rules apply to it, ie. as ephemeral zNode, it's destroyed once consumer goes down and the registration process is made automatically by the consumer.

P64 - 65

What is the Architecture of ZooKeeper?

- ZooKeeper is a distributed application on its own while being a coordination service for distributed systems.
- It has a simple client-server model in which clients are nodes (i.e. machines) and servers are nodes.
- As a function, ZooKeper Clients make use of the services and servers provides the services.
- Applications make calls to ZooKeeper through a client library.
- The client library handles the interaction with ZooKeeper servers here.

P66

Design Goals of Zookeeper Architecture

There were some motives behind the design of Zookeeper Architecture:

- ZooKeeper architecture must be able to tolerate failures.
- Also, it must be in the position to recover from correlated recoverable failures (power outages).
- Most importantly it must be correct or easy to implement correctly.
- Additionally, it must be fast along with high throughput and low latency.

a. ZooKeeper is simple

While working on ZooKeeper, all distributed processes can coordinate with each other. This coordination is possible through a shared hierarchical namespace. However, it is organized as same as the standard file system. Here the namespaces which consist of data registers, what we call as znodes, in ZooKeeper parlance. Though, these are as same as files and directories. In addition, ZooKeeper data keeps in-memory, due to that it achieves high throughput as well as low latency numbers.

b. ZooKeeper is replicated

Apache ZooKeeper itself is intended to be replicated over a set of hosts called an ensemble, as same as distributed processes it coordinates.

c. How is the order beneficial?

In order to implement higher-level abstractions (synchronization primitives, Subsequent operations) usage of the order is required.

d. ZooKeeper is fast

Especially, in “read-dominant” workloads, ZooKeeper works very fast.

P67

Data Model in ZooKeeper

As same as a *standard file system*, the *namespace* provided by ZooKeeper. Basically, a sequence of path elements which separates by a slash (/) is what we call a name. In ZooKeeper’s namespace, a path identifies every node. Moreover, in a ZooKeeper namespace, each node can have data associated with it and its children. As same as a file-system which permits a file to also be a directory.

At each ZNode in a namespace, read and write of data is automatically. That says, here Reads get all the data bytes which correspond with a ZNode whereas write replaces all the data. In addition, there is an *Access Control List (ACL)* with each node that restricts everybody's work.
Additionally, within each of the ZooKeeper servers, the ZNode hierarchy is stored in memory. Basically, that helps for quick responses to reads from the clients. This hierarchy can offer reliability, availability, and coordination to our application that's why we must use it as a storage mechanism for the small amount of data.

Information conveyed in ZooKeeper Architecture

The absence of data often conveys important information about a ZNode in ZooKeeper Architecture.

1. To all ZNodes representing a worker available in the system, the /workers ZNode is the parent ZNode. So, its ZNode should be removed from /workers, if a worker becomes unavailable.
2. Since waiting for workers to execute tasks, the /tasks ZNode is the parent of all the tasks created. In order to represent new tasks and wait for ZNodes representing the status of the task, clients of the master-worker application add new ZNodes as children of /tasks.
3. By representing an assignment of a task to a worker, the /assign ZNode is the parent of all ZNodes. Also, it adds a child ZNode to / assigns, when a master assigns a task to a worker.

P68

What is Zookeeper ZNodes?

The term *ZNode is referred to every node in a ZooKeeper tree*. The *main purpose of the Znode is to maintain a stat structure*. However, stat structure includes version numbers for data changes and ACL changes. Also, a stat structure includes timestamps in it. Hence, timestamps and the version number together permits ZooKeeper to validate the cache as well as to coordinate updates. Although make sure, the version number increases, each time a ZNode's data changes. For instance, a Client also receives the version of the data, whenever it retrieves data. Though, a client must supply the version of the data of the ZNode it is changing, when a client performs an update or a delete. Because the update will fail if the version it supplies doesn't match the actual version of the data.

Do you know about ZooKeeper Leader Election

In addition, note that the word node can refer to many in a distributed application engineering. for example, a generic host machine, a server, a member of an ensemble, a client process, etc. Basically, ZNodes refer to the data nodes, in the

ZooKeeper documentation. Along with it, the Servers to refer to machines which make up the ZooKeeper service, similarly, quorum peers refer to the servers which make up an ensemble; in the same way, a client refers to any host or process that uses a ZooKeeper service.
Although, we can say, the main entity that a programmer access are ZNodes.

3. Characteristics of Zookeeper ZNodes

Further, there are several characteristics of Zookeeper Znodes, such as:

a. Watches

It is possible for clients to set watches on ZNodes. So, when we make changes to that ZNode, as a result, it triggers the watch and then further clear the watch. Also, ZooKeeper sends the client a notification, when a watch triggers.

b. Data Access

At each ZNode, the data which is stored in a namespace is read and written atomically. Basically, Read process get all the data bytes which are associated with a ZNode. Whereas, the writing process replaces all the data. In addition, there is an Access Control List (ACL) of each node, so that restricts the function of all.
Let's revise ZooKeeper CLI

c. Ephemeral Nodes

Well, we can say, there is a slight notion of ephemeral nodes in ZooKeeper. As long as the session which creates the ZNode is active, until that time these ZNodes exists. Similarly, the ZNode is deleted, when the session ends. Thus, ephemeral ZNodes are not allowed to have children, because of this behavior only.

d. Sequence Nodes – Unique Naming

We can also request that ZooKeeper append a monotonically increasing counter to the end of the path while creating a ZNode. Well, it is a unique counter to the parent ZNode. The format of the counter is %010d — basically, it is 10 digits with 0 (zero) padding. However, to simplify sorting, the counter is formatted in this way, like "<path>0000000001"
Also, note that maintained by the parent node, the counter used to store the next sequence number is a signed int (4bytes), and also the counter will overflow while it is incremented beyond 2147483647.
So, this was all about ZooKeeper ZNodes. Hope you like our explanation.

ZooKeeper Architecture – Modes for ZNodes
We also need to specify a mode, when creating a new ZNode in ZooKeeper architecture. Because different modes explain the behavior of the ZNode:
Let's explore ZooKeeper benefits and limitations

a. Persistent and Ephemeral znodes
A ZNode can be of any type: either a *persistent ZNode* or an *ephemeral ZNode*. Basically, only through a call to delete, we can delete a persistent ZNode/path. And, in contrast, if the client that created it crashes or simply closes its connection to ZooKeeper, an ephemeral ZNode deletes.
Generally, the ZNode stores some data on behalf of an application. Even after its creator is no longer part of the system, and it is a need to preserve its data, in that case, Persistent ZNodes are useful. Whereas, when some aspect of the application that must exist only while the session of its creator is valid, Ephemeral ZNodes convey information about that.
b. Sequential ZNodes
These ZNodes have a unique, monotonically increasing integer which we further use to create the ZNode. In other words, these offer an easy way to create ZNodes with unique names. Also, offer a way to easily see the creation order of ZNodes.

P69

There is a version number that associates with every ZNode. Further, that number increases every time its data changes. Especially, at the time when multiple ZooKeeper clients might be trying to perform operations over the same ZNode, the use of versions is important.

Writing (or Re-Writing) to a Znode

We use `setData` for writing to a znode. This method takes in 3 parameters: Path [String] as always, data [byte Array] that will overwrite the pre-existing data, and the version [int].
We have a new parameter (but fairly self-explanatory), which is version. Every time the znode gets updated, it makes sense to update its version. Because of this, if you try to pass in the integer value that is not a current version, it will throw a `BadVersionException`.

P70

2. What is ZooKeeper Watches?

In ZooKeeper, all of the read operations – getData(), getChildren(), and exists() – have the option of setting a watch as a side effect. Defining ZooKeeper Watches, when the data for which the watch was set changes, a watch event (one-time trigger), sent to the client which set the watch. However, in this definition of a watch, there are three key points to consider, such as:

Do you know about ZooKeeper Data Models

a. One-time trigger

When any change occurs in data, one watch event will be sent to the client. Let's say the client will get a watch event for /znode1 if a client does a getData("/znode1", true) and later the data for /znode1 is changed or deleted. However, it will not send any watch event unless the client performs another read which sets a new watch, if /znode1 changes again.

b. Sent to the client

It simply means that an event is on the way to the client. However, it may not reach the client somehow, before the successful return code to the change operation reaches the client which initiated the change. Asynchronously, ZooKeeper Watches send to ZooKeeper watchers. Also, a client will never see a change for which it has set a watch until it first sees the watch event, this is an ordering guarantee offered by ZooKeeper.

However, sometimes, the network may delay or also other factors may cause different clients to see watches and further return codes from updates at different times. Though we can say, different clients will have a consistent order is the key point of it.

Test your ZooKeeper Learning

c. The data for which the watch was set

It says in different ways we can change a node. Basically, that makes us think of ZooKeeper as maintaining two lists of watches. Those are Data ZooKeeper Watches and Child ZooKeeper Watches. To set data watches, getData() and exists() and to set child watches, getChildren().

However, both getData() and exists() gets the information about the node's data. And, getChildren() gets the list of children. Hence, successful setData() will trigger data watches to set the ZNode.

Locally, ZooKeeper servers which are connected to the client maintain the watches. Though it permits in order to set

watches they are must lightweight. Moreover, the watch will trigger for any session events, when a client connects to a new server. However, while disconnected from a server, watches will not receive.
But at the time when the client reconnects, any previously registered watches will trigger as well as reregistered as per requirement. Also, we can say this all occurs transparently. Moreover, the watch can also be missed if the Znode is created and deleted while disconnected, then a watch for the existence of a ZNode not yet created will be missed.
Let's revise ZooKeeper Barriers

3. What ZooKeeper Guarantees about Watches?

ZooKeeper maintains various guarantees, with regard to watches:

- Basically, with respect to other events, other watches, and asynchronous replies, watches get in order. In addition, the ZooKeeper client libraries ensure that if everything gets dispatch in order or not.
- Before seeing the new data that corresponds to that znode, a client will see a watch event for a **znode** it is watching.
- ZooKeeper Watch events order corresponds to the order of the updates as seen by the ZooKeeper service.

4. Things to Remember about Watches

Let's discuss ZooKeeper Workflow

- We must set another watch if we get a watch event and we want to get notified of future changes. It is because watches are one time triggers.
- Though, it is not possible for us to reliably see every change that happens to a node in ZooKeeper. It is due to latency between getting the event as well as sending a new request to get a watch. And also because watches are one time triggers. So, where the ZNode changes multiple times between getting the event and setting the watch again, just prepare yourself to handle the case.
- For a given notification, a watch object, or function/context pair, will only trigger once.
- Moreover, we will not get any watches until the reestablishment of connection, while we disconnect from a server.

So, this was all in Zookeeper Watches. Hope you like our explanation.

P71

What is Quorum?

Minimum number of nodes in cluster that must be online and be able to communicate with each other. If any additional node failure occurs beyond this threshold, cluster will stop running.

What value should we choose for Quorum?

More than half of the number of nodes in cluster. (N/2 + 1) where N is total number of nodes in cluster

Basically, a ZooKeeper replicates its data a tree across all servers in the ensemble, in quorum mode. Yet, the delays might be unacceptable, if a client had to wait for every server to store its data before continuing. Generally, a quorum is the minimum number of legislators needs to be present for a vote, in public administration. In Zookeeper also, it is the minimum number of servers that have to be running and available in order, to make Zookeeper work.

P72

Memory
Basically, ZooKeeper is not a memory intensive application when handling only data stored by Kafka. Make sure, a minimum of 8 GB of RAM should be there for ZooKeeper use, in a typical production use case.

CPU
As a Kafka metadata, ZooKeeper store does not heavily consume CPU resources. ZooKeeper also offers a latency sensitive function. That implies we must consider providing a dedicated CPU core to ensure context switching is not an issue if it must compete for CPU with other processes.
Let’s revise Apache Kafka Security | Need and Components of Kafka

Disks
In order to maintain a healthy ZooKeeper cluster, Disk performance is very essential. To perform optimally, we recommend using Solid state drives (SSD) as ZooKeeper must have low latency disk writes.
+++ storing logs separately,
+++ isolating the ZooKeeper process,
+++ and **disabling swaps will also reduce latency.**

P73

A. Operating System

When the ZooKeeper service will start to struggle, the underlying OS metrics can help predict. We should monitor:

i. Number of open file handles
This should be done system-wide and for the user running the ZooKeeper process. Values should be considered with respect to the maximum allowed number of open file handles. It also opens and closes connections often, and moreover, it needs an available pool of file handles to choose from.
Let’s discuss Kafka Client

ii. Network bandwidth usage
ZooKeeper is sensitive to timeouts caused by network latency just because it keeps track of state. If somehow the network bandwidth is saturated then only it is possible that we may experience hard to explain timeouts with client sessions, although that results in making Kafka cluster less reliable.

B. “Four Letter Words”
Basically, ZooKeeper only response to a set of commands, each one is of four letters. To run the commands we must send a message (via netcat or telnet) to the ZooKeeper client port. For example echo stat | nc localhost 2181 would return the output of the STAT command to stdout.

C. JMX Monitoring
JMX metrics that are important to monitor:
NumAliveConnections
OutstandingRequests
AvgRequestLatency
MaxRequestLatency
HeapMemoryUsage (Java built-in)
Moreover, by the SessionExpireListener, Kafka, tracks the number of relevant ZooKeeper events. Basically, it is monitored to ensure the health of ZooKeeper-Kafka interactions:
Read Apache Kafka Architecture and its fundamental concepts
ZooKeeperAuthFailuresPerSec (secure environments only)
ZooKeeperDisconnectsPerSec
ZooKeeperExpiresPerSec
ZooKeeperReadOnlyConnectsPerSec
ZooKeeperSaslAuthenticationsPerSec (secure environments only)
ZooKeeperSyncConnectsPerSec

P74

Apache Reference:

- Hardware

    - We are using dual quad-core Intel Xeon machines with 24GB of memory.
- **CPU**: Unless SSL and log compression are required, a powerful CPU isn't needed for Kafka. Also, the more cores used, the better the parallelization. In most scenarios, where compression isn't a factor, the LZ4 codec should be used to provide the best performance.
- **RAM**: In most cases, Kafka can run optimally with 6 GB of RAM for heap space. For especially heavy production loads, use machines with 32 GB or more. Extra RAM will be used to bolster OS page cache and improve client throughput. While Kafka can run with less RAM, its ability to handle load is hampered when less memory is available.
- OS
    - Kafka should run well on any unix system and has been tested on Linux and Solaris.
    - We have seen a few issues running on Windows and Windows is not currently a well supported platform though we would be happy to change that.
- Filesystem
    - Kafka uses regular files on disk, and as such it has no hard dependency on a specific filesystem. The two filesystems which have the most usage, however, are EXT4 and XFS. Historically, EXT4 has had more usage, but recent improvements to the XFS filesystem have shown it to have better performance characteristics for Kafka's workload with no compromise in stability.
- **Disk**: Kafka thrives when using multiple drives in a RAID setup. SSDs don't deliver much of an advantage due to Kafka's sequential disk I/O paradigm, and NAS should not be used.

- **Network and File System**: XFS is recommended, as is keeping your cluster at a single data center if circumstances allow. Also, deliver as much network bandwidth as possible.

P75

OS-level tuning [File descriptor limits, Max socket buffer, Maximum number of memory map]

The most important **producer** configurations are: [Acks – compression - batch size]

The most important **consumer** configuration is the **fetch size.**

Java Version
Java 8 and Java 11 are supported. Java 11 performs significantly better if TLS is enabled, so it is highly recommended (it also includes a number of other performance improvements: G1GC, CRC32C, Compact Strings, Thread-Local Handshakes and more). From a security perspective, we recommend the latest released patch version as older freely available versions have disclosed security vulnerabilities.

P76

Typical arguments for running Kafka with OpenJDK-based Java implementations (including Oracle JDK) are:

```
-Xmx6g -Xms6g -XX:MetaspaceSize=96m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M
```

```
-XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:+ExplicitGCInvokesConcurrent
```

For reference, here are the stats for one of LinkedIn's busiest clusters (at peak) that uses said Java arguments:

- 60 brokers
- 50k partitions (replication factor 2)
- 800k messages/sec in
- 300 MB/sec inbound, 1 GB/sec+ outbound

All of the brokers in that cluster have a 90% GC pause time of about 21ms with less than 1 young GC per second.

P78

Horizontally scalable

Let's define the term *vertical scalability* first. Say, for instance, you have a traditional database server that's 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:

- There are limits defined by the hardware. You cannot scale upwards indefinitely
- It usually requires downtime, something which big corporations can't afford

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

Fault-tolerant

Something that emerges in non distributed systems is 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 two of the nodes are down. It's worth noting that fault tolerance is at a direct trade-off with performance, as in the more fault-tolerant your system is, the less performant it is.

Scale-out architecture with Kafka:

Partition: Splits a topic into multiple partitions and increasing partitions is a mechanism of scaling.

Distribution: Cluster can have one or more brokers and these brokers can be increased to achieve scaling.

Replication: Similar to partitions, multiple replication of a message is there for fault-tolerance and this aspect also brings in scalability in Kafka.

Scaling: Each consumer reads a message from a single partition (of a topic) and to scale out we add more consumers and the newly added consumers read the message from new partition (one consumer cannot read from the same partition; this is a rule) as shown in this figure.

P80

What is fault tolerance?

In the previous session, we learned that Kafka is a distributed system and it works on a cluster of computers. Most of the time, Kafka will spread your data in partitions over various systems in the cluster. So, if one or two systems in a cluster fail, what will happen to your data? Will you be able to read it?
Probably not. That's a fault. Can we tolerate it?
The term fault tolerance is very common in distributed systems. It means, making your data available even in the case of some failures.
How to do it?
One simple solution is to make multiple copies of the data and keep it on separate systems. So if you have three copies of a partition, and Kafka stores them on three different machines, you should be able to avoid two failures. Since you have three copies on three different systems, even if two of them fails, you can still read your data from the third system.

Replication Factor

There is a particular term used for making multiple copies. We call it replication factor. So, if I say, replication factor is three, that means, I am maintaining three copies of my partition. If I say replication factor is two, that means we are keeping two copies of a partition. The replication factor of three is a reasonable number. You can even set it to higher if your data is supercritical or you are using cheap machines.
So, Kafka implements fault tolerance by applying replication to the partitions. We can define replication factor at the Topic level. We don't set a replication factor of partitions, we set it for a Topic, and it applies to all partitions within the Topic.

Apache Kafka is a distributed system, and distributed systems are subject to multiple types of faults. Some of the classic cases are:

1. A broker stops working, becomes unresponsive, and cannot be accessed.
2. Data is stored on disks, the disk fails, and then the data cannot be accessed.
3. Suppose that there are multiple brokers in a cluster. Each broker is a leader of more than one partition. If one of those brokers fails or is inaccessible, then it will result in loss of data.

In these scenarios, ZooKeeper comes to the rescue. The moment ZooKeeper realizes that one of the brokers is down, it performs the following actions:

1. It will find another broker to take the place of the failed broker.
2. It will update the metadata used for work distribution for producers and consumers in order to make sure that processes continue.

Once ZooKeeper has performed these two steps, the publishing and the consumption of the messages will continue as normal. The challenge here is with the failed broker that still holds data. Unless some provision is made to replicate the data somewhere else, that data will be lost.

P81

Sometimes being available is necessary for data safety. You need to think about:

- Can my publisher simply return an error upstream and the upstream service or user can retry later?
- Can my publisher persist the message locally or to a database so it can retry later?

If the answer is no then optimizing for availability may end up being better for data safety. Either way you will end up losing data but you may lose less data optimizing for availability than refusing writes. So it is a balancing act and the decision depends on your situation.

Consumer failures

Consumers in a consumer group can share processing load. If a consumer unexpectedly fails, Kafka can detect the failure and rebalance the partitions amongst the remaining consumers in the consumer group. The consumer failures can be hard failures (for example, `SIGKILL`) or soft failures (for example, expired session timeouts). These failures can be detected either when consumers fail to send heartbeats or when they fail to send `poll()` calls. The consumer liveness is maintained with a heartbeat, now in a background thread since KIP-62, and the configuration parameter `session.timeout.ms` dictates the timeout used to detect failed heartbeats. Increase the session timeout to take into account potential network delays and to avoid soft failures. Soft failures occur most commonly in two cases: when `poll()` returns a batch of messages that take too long to process or when a JVM GC pause takes too long. If you have a `poll()` loop that spends much time processing messages, you can do one of the following:

- Increase the upper bound on the amount of time that a consumer can be idle before fetching more records with `max.poll.interval.ms`.
- Reduce the maximum size of batches the `max.poll.records` configuration parameter returns.

Although higher session timeouts increase the amount of time to detect and recover from a consumer failure, failed client incidents are less likely than network issues.

P82

Clients can be given multiple brokers that they can connect to, in the bootstrap.servers producer and consumer configs. The idea is that even if one node goes down, the client has multiple nodes it knows about and can open a connection to them. The bootstrap servers might not be the leaders of the partitions the client needs, instead the boostrap servers are a bridgehead. The client can ask them which node hosts the leader of the partition they want to read/write to.

P83

So far we haven't covered how a cluster can know when a broker has failed, or how leadership election occurs. In order to cover how Kafka deals with network partitions first we'll need to understand Kafka's consensus architecture.

Each cluster of Kafka nodes is deployed alongside a Zookeeper cluster. Zookeeper is a distributed service that allows a distributed system to attain consensus around some given state. It is distributed itself and is has chosen consistency over availability. A majority of Zookeeper nodes are required in order to accept reads and writes.

Zookeeper is responsible for storing state about the Kafka cluster:

- The list of topics, the partitions, configuration, current leader replicas, preferred replicas.
- The cluster members. Each broker sends a heartbeat to the Zookeeper cluster, when Zookeeper fails to receive a heartbeat after period of time Zookeeper assumes the broker has failed or otherwise unavailable.
- Electing the controller node which includes controller node fail-over when the controller dies.

The controller node is one of the Kafka brokers which has the responsibility of electing replica leaders. Zookeeper sends the Controller notifications about cluster membership and topic changes and the Controller must act on those changes.

For example, when a new topic is created with 10 partitions and a replication factor of three. The controller must elect one leader per partition, trying to distribute the leaders optimally across the brokers.

For each partition it:

- updates Zookeeper with the ISR and leader
- sends a LeaderAndISRCommand to each broker that hosts a replica of that partition, informing the brokers of the ISR and leader.

When a Kafka broker dies that hosts a replica leader, Zookeeper sends a notification to the Controller and it will elect a new leader. Again, the Controller updates Zookeeper first, then sends a command to each hosting broker notifying them of the leadership change.

Each leader is responsible for maintaining the ISR. It uses the replica.lag.time.max.ms to determine membership of the ISR. When the ISR changes, the leader updates Zookeeper of the change.

Zookeeper is always updated of any change in state so that in the case of a fail-over, that the new leader can smoothly transition into leadership.

P84

The Replication Protocol

Understanding the details of replication helps us understand potential data loss scenarios better.

Fetch requests, Log End Offset (LEO) and the Highwater Mark (HW)

We have covered that followers are periodically sending fetch requests to the leader. The default interval is 500ms. This differs from RabbitMQ in that with RabbitMQ, the replication is not initiated by the queue mirror but by the master. The master pushes changes to the mirrors.

The leader and each follower stores a Log End Offset (LEO) and Highwater Mark (HW). The LEO is the last message offset the replica has locally and the HW is the last committed offset. Remember that to be committed, a message must have been persisted to each replica in the ISR. That means that the LEO is likely a little ahead of the HW.

When the leader receives a message, it persists it locally. A follower makes a fetch request, sending its own LEO. The leader then sends a batch of messages starting from that LEO and also sends the current HW. When the leader knows

that all replicas have persisted a message at a given offset, it advances the HW. Only the leader is able to advance the HW and it lets the followers all know the current value in the fetch responses. This means that the followers may be lagging behind the leader regarding the messages but also regarding knowing the current HW. Consumers are only delivered messages up to the current HW.

Note that "persisted" means written to memory, not disk. For performance, Kafka fsyncs to disk on an interval. RabbitMQ also writes to disk periodically, but the difference is that RabbitMQ will only send a publisher confirm once the master and all mirrors have written the message to disk. Kafka has made the decision to acknowledge once a message is in memory for performance reasons. Kafka is making the bet that redundancy will make up for the risk of storing acknowledged messages in memory only for a short period of time.

P85

Leader Fail-Over

When a leader fails, the Controller is notified by Zookeeper and elects a new leader replica. The new leader will make the new HW its current LEO. The followers will then be informed of the new leader. Depending on the version of Kafka, each follower will:

- truncate its local log to the HW it knows about and makes a fetch request to the new leader from that offset
- make a request to the leader to know the HW at the time of it’s election to leader, then truncate its log to that offset. Then start making periodic fetch requests, starting at the offset.

The reason that a follower partition may need to truncate its log after leader election is that:

- When a leader fail-over occurs, the first follower in the ISR to register itself to Zookeeper wins the leader election and becomes the leader. Each follower in the ISR, while being "in-sync" may or may not be fully caught up with the former leader. It is possible that the follower that gets elected is not the most caught up. Kafka ensures that there is no divergence between replicas. So to avoid divergence, each follower must truncate to the HW of the new leader at the time of its election. This is another reason why acks=all is so important if you must have consistency.
- Messages are written to disk periodically. If all nodes of a cluster failed simultaneously, different replicas will have persisted to disk up to a different offset. It is entirely possible that when the brokers come back online again, the new leader that gets elected could be behind its followers because it made its last fsync further in the past than its peers.

P86

Network Partitions

Kafka has a more complex set of behaviors when a cluster suffers a network partition. But Kafka was built from day one to run as a cluster and is well thought out when it comes to network partitions.

Below are a few different network partition scenarios:

- Scenario 1: A follower cannot see the leader, but can still see Zookeeper
- Scenario 2: A leader cannot see any of its followers, but can still see Zookeeper
- Scenario 3: A follower can see the leader, but cannot see Zookeeper
- Scenario 4: A leader can see its followers, but cannot see Zookeeper
- Scenario 5: A follower is completely partitioned from both the other Kafka nodes and Zookeeper
- Scenario 6: A leader is completely partitioned from both the other Kafka nodes and Zookeeper

Each of the above will exhibit different behaviors.

P87

Scenario 1: A follower cannot see the leader, but can still see Zookeeper

A network partition separates broker 3 from brokers 1 and 2, but not from Zookeeper. Broker 3 is no longer able to send fetch requests and after *replica.lag.time.max.ms* is removed from the ISR and does not contribute to message commits. As soon as the partition is resolved it will resume fetch requests and rejoin the ISR when caught up with the leader. Zookeeper will continue to receive heartbeats throughout and will assume the broker to be alive and well throughout.

There is no split brain or paused node, instead it suffers reduced redundancy.

P88

Scenario 2: A leader cannot see any of its followers, but can still see Zookeeper

The network partition separates the leader partition from its followers, but the broker can still see Zookeeper. Just like with Scenario 1, the ISR shrinks but this time it shrinks to only the leader as the followers all cease sending fetch requests. Again, there is no split-brain. Instead there is a loss of redundancy for new messages until the partition is resolved. Zookeeper will continue to receive heartbeats throughout and will assume the broker to be alive and well throughout.

P89

Scenario 3: A follower can see the leader, but cannot see Zookeeper

A follower is partitioned from Zookeeper but not from the broker with the leader. The result is that the follower continues to make fetch requests and continues to be a member of the ISR. Zookeeper will no longer receive heartbeats and will assume the broker to be dead, but as it is only a follower there are no repercussions.

P90

Scenario 4: A leader can see its followers, but cannot see Zookeeper

The leader is partitioned from Zookeeper but not from the brokers with the followers.

After a short while Zookeeper will mark the broker as dead and notify the Controller. The Controller will elect a follower as the new leader. However the original leader will continue to think it is the leader and will continue to accept writes with acks=1. The followers will no longer be sending fetch requests to the original leader and so the original leader will presume them dead and attempt to shrink the ISR to itself. But because it has no connectivity to Zookeeper it won't be able to and at that point it will refuse more writes.

Acks=all messages will not be acknowledged because at first the ISR includes all replicas, but they will not acknowledge receipt of the messages. When the original leader attempts to remove them from the ISR it will fail to do so and stop accepting any messages at all.

P91

Continue… Scenario 4. The leader on Broker 1 becomes a follower after the network partition is resolved.

Clients soon detect the leadership change and start writing to the new leader. Once the network partition is resolved the original leader will see that it is no longer the leader and will truncate its log to the HW that the new leader had when the fail-over occurred. This is to avoid divergence of their logs. It will then start sending fetch requests to the new leader. Any writes in the original leader that had not been replicated to the new leader are lost. That is, the messages acknowledged by the original leader during those seconds when there were two leaders, will be lost.

P92

Scenario 5: A follower is completely partitioned from both the other Kafka nodes and Zookeeper

A follower is completely isolated from both its peer Kafka brokers and Zookeeper. It will simply be removed the ISR until the network partition is resolved and it can catch up again.

P93

Scenario 6: A leader is completely partitioned from both the other Kafka nodes and Zookeeper

A leader is completely isolated from its followers, the Controller and from Zookeeper. It will continue to accept writes with acks=1 for a short period.

After *replica.lag.time.max.ms* has passed without fetch requests it will try to shrink the ISR to itself but will be unable to do so as it cannot talk to Zookeeper and it will stop accepting writes.

Meanwhile, Zookeeper will have marked the isolated broker as dead and the Controller node will have elected a new leader.

The original leader may accept writes for a few seconds but then stop accepting any messages. Clients update themselves every 60 seconds with the latest meta data. They will be informed of the leader change and start writing to the new leader.

Any acknowledged writes made to the original leader since the network partition began will be lost. Once the network partition is resolved the original leader will discover it is no longer the leader via Zookeeper. It will then truncate its log to the HW of the new leader at the time of the election and start fetch requests as a follower.

This is a case where a partition can be in split-brain for a short period, though only if acks=1 and min.insync.replicas is 1. The split-brain is automatically ended either when the network partition is resolved and the original leader realizes it is no longer the leader, or all clients realize the leader has changed and start writing to the new leader - whichever happens first. Either way, some message loss will occur but only with acks=1.

There is also a variant of this scenario where just before the network partition, the followers fell behind and the leader shrunk the ISR to itself. Then the network partition isolates the leader. A new leader is elected but the original leader continues to accept writes, even acks=all because the ISR is already only itself. These writes will be lost when the network partition is resolved. So to avoid this variant the only solution is to use min.insync.replicas = 2.

P94

Continue…. Scenario 6: The original leader becomes a follower after the network partition is resolved

P95

Scenarios Conclusions

We see that network partitions that affect followers do not result in message loss, just reduced redundancy for the duration of the network partition. This of course can lead to data loss if one or more nodes are lost.

Network partitions that isolate leaders from Zookeeper can end up with message loss for messages with acks=1. Not being able to see Zookeeper causes short duration split-brains where we have two leaders. The remedy for this is to use acks=all.

Message Loss Recap

Let's list the ways in which you can lose data with Kafka:

- Any leader fail-over where messages were acknowledged with acks=1
- Any unclean fail-over (to a follower outside of the ISR), even with acks=all

- A leader isolated from Zookeeper receiving messages with acks=1
- A fully isolated leader whose ISR was already shrunk to itself, receving any message, even acks=all. This is true only if min.insync.replicas=1.
- Simultaneous failures of all nodes of a partition. Because messages are acknowledged once in memory, some messages may not yet have been written to disk. When the nodes come back up some messages may have been lost.

Unclean fail-overs can be avoided by either disabling them or ensuring that redundancy is always at least two. The most durable configuration is a combination of acks=all and min.insync.replicas greater than 1.

P97

Let's go back to some basics

Apache Kafka is a streaming platform based on a distributed publish/subscribe pattern. First, processes called ***producers*** send messages into ***topics***, which are managed and stored by a cluster of ***brokers***. Then, processes called ***consumers*** subscribe to these topics for fetching and processing published messages.

A topic is distributed across a number of brokers so that each broker manages subsets of messages for each topic - these subsets are called ***partitions.*** The number of partitions is

defined when a topic is created and can be increased over time (but be careful with that operation).

What is important to understand is that a partition is actually the **unit of parallelism** for Kafka's producers and consumers.

On the producer side, the partitions allow writing messages in parallel. If a message is published with a key, then, by default, the producer will hash the given key to determine the destination partition. This provides a guarantee that all messages with the same key will be sent to the same partition. In addition, a consumer will have the guarantee of getting messages delivered in order for that partition.

On the consumer side, the number of partitions for a topic bounds the maximum number of active consumers within a ***consumer group.*** A consumer group is the mechanism provided by Kafka to group multiple consumer clients, into one logical group, in order to load balance the consumption of partitions. Kafka provides the guarantee that a topic-partition is assigned to only one consumer within a group.

For example, the illustration below depicts a consumer group named A with three consumers.

If a consumer leaves the group after a controlled shutdown or crashes then all its partitions will be reassigned automatically among other consumers. In the same way, if a consumer (re)join an existing group then all partitions will be also rebalanced between the group members.

The ability of consumers clients to cooperate within a dynamic group is made possible by the use of the so-called **Kafka Rebalance Protocol.**

P98

The Rebalance Protocol, in a Nutshell

First, let's give a definition of the meaning of the term "**rebalance**" in the context of Apache Kafka.

Rebalance/Rebalancing: the procedure that is followed by a number of distributed processes that use Kafka clients and/or the Kafka coordinator to form a common group

and distribute a set of resources among the members of the group (source : Incremental Cooperative Rebalancing: Support and Policies).

This definition above actually makes no reference to the notion of consumers or partitions. Instead, it uses a concept of **members** and **resources**. The main reason for that is because the rebalance protocol is not only limited to manage consumers but can also be used to coordinate any group of processes.

P99

Here are some usages of the protocol rebalance:

- **Confluent Schema Registry** relies on rebalancing to elect a leader node.
- **Kafka Connect** uses it to distribute tasks and connectors among the workers.
- **Kafka Streams** uses it to assign tasks and partitions to the application streams instances.

In addition, what is really important to understand is that rebalance mechanism is actually structured around two protocols: **Group Membership Protocol** and **Client Embedded Protocol**.

The Group Membership Protocol, as its name suggests, this protocol is in charge of the coordination of members within a group. The clients participating in a group will execute a sequence of requests/responses with a Kafka broker that acts as **coordinator**.

The second protocol is executed on the client side and allows extending the first one by being embedded in it. For example, the protocol used by consumers will assign topic-partition to members.
Now that we have a better understanding of what the rebalance protocol is, let's illustrate its implementation for assigning partitions in a consumer group.

P100

When a consumer starts, it sends a first `FindCoordinator` request to obtain the Kafka broker coordinator which is responsible for its group. Then, it initiates the rebalance protocol by sending a `JoinGroup` request.
As we can see, the `JoinGroup` contains some consumer client configuration such as the `session.timeout.ms` and the `max.poll.interval.ms`. These properties are used by the coordinator to kick members out of the group if they don't respond.
In addition, the request also contains two very important fields: the list of client protocols, supported by the members, and metadata that will be used for executing one of the embedded client protocols. In our case, the client-protocols are the list of partition assignors configured for the consumer (i.e : `partition.assignment.strategy)`. Metadata contains the list of topics the consumer has subscribed to.

P101

The `JoinGroup` acts as a barrier, meaning that the coordinator doesn't send responses as long as all consumer requests are not received (i.e `group.initial.rebalance.delay.ms)` or rebalance timeout is reached.

The first consumer, within the group, receives the list of active members and the selected assignment strategy and acts as **the group leader** while others receive an empty response. The group leader is responsible for executing the partitions assignments locally.

P102

Next, all members send a `SyncGroup` request to the coordinator. The group leader attached the computed assignments while others simply respond with an empty request.

P103

Once the coordinator responds to all `SyncGroup` requests, each consumer receives their assigned partitions, invokes the `onPartitionsAssignedMethod` on the configured listener and, then starts fetching messages.

P104

Heartbeat

Last but not least, each consumer periodically sends a `Heatbeat` request to the broker coordinator to keep its session alive (see : `heartbeat.interval.ms`).
If a rebalance is in progress, the coordinator uses the `Heatbeat` response to indicate to consumers that they need to rejoin the group.
So far so good, but as you should know in a real-life situation and more especially in a distributed system, failures will happen. Hardware can fail. The network or a consumer can have transient failures. Unfortunately, for all these situations a rebalance can also be triggered.

P105

The first limitation of the rebalance protocol is that we cannot simply rebalance one member without stopping the whole group (*stop-the-world effect*).
For example, let's properly stop one of our instances. In this first rebalance scenario, the consumer will send a LeaveGroup request to the coordinator, before stopping.

P106

Remaining consumers will be notified that a rebalance must be performed on the next `Heartbeat` and will initiate a new `JoinGroup/SyncGroup` round-trip in order to reassign partitions.
During the entire rebalancing process, i.e. as long as the partitions are not reassigned, consumers no longer process any data. By default, the rebalance timeout is fixed to 5 minutes which can be a very long period during which the increasing consumer-lag can become an issue.

P107

But what would happen, if, for example, the consumer was just restarting after a transient failure ? Well, the consumer, while rejoining the group, will trigger a new rebalance causing all consumers to be stopped (once again).
Another reason that can lead to a restart of a consumer is a rolling upgrade of the group.
This scenario is unfortunately disastrous for the consumption group. Indeed, with a group of three consumers, such operation will trigger 6 rebalances that could potentially have a significant impact on messages processing.

P108

Finally, a common problem when running Kafka consumers, in Java, is either missing a heartbeat request, due to a network outage or a long GC pause, or not invoking the method `KafkaConsumer#poll()`, periodically, due to an excessive processing time. In the first case, the coordinator is not receiving a heartbeat for more than`session.timeout.ms` milliseconds and considers the consumer dead. In the second one, the time needed for processing polled records is superior to `max.poll.inteval.ms`.

P109

Static Membership

To reduce consumer rebalances due to transient failures, Apache Kafka 2.3 introduces the concept of Static Membership with the KIP-345.

The main idea behind static membership is that each consumer instance is attached to a unique identifier configured with `group.instance.id`. The membership protocol has been extended so that ids are propagated to the broker coordinator through the `JoinGroup` request.

If a consumer is restarted or killed due to a transient failure, the broker coordinator will not inform other consumers that a rebalance is necessary until `session.timeout.ms` is reached. One reason for that is that consumers will not send `LeaveGroup` request when they are stopped.

P110

When the consumer will finally rejoin the group, the broker coordinator will return the cached assignment back to it, without doing any rebalance.

When using static membership, it's recommended to increase the consumer property `session.timeout.ms` large enough so that the broker coordinator will not trigger rebalance too frequently.

On the one hand, static membership can be very useful for limiting the number of undesirable rebalances and thus minimizing stop-the-world effect. On the other hand, this has the disadvantage of increasing the unavailability of partitions because the coordinating broker may only detect a failing consumer after a few minutes (depending on `session.timeout.ms`). Unfortunately, this is the eternal trade-off between availability and fault-tolerance you have to make in a distributed system.

P111

Incremental Cooperative Rebalancing

As of version 2.3, Apache Kafka also introduces new embedded protocols to improve the resource availability of each member while minimizing stop-the-world effect.
The basic idea behind these new protocols, is to perform rebalancing incrementally and in cooperation — In other words, it means executing multiple rebalance rounds rather than a global one.
Incremental Cooperative Rebalancing was first implemented for Kafka Connect through the KIP-415 (partially implemented in Kafka 2.3). Moreover, it will be available for streams and consumers from Kafka 2.4 through the KIP-429.

P112

KIP - Rebalancing Protocols
KIP: Kafka Improvement Proposals

* KIP-345: Introduce static membership protocol to reduce consumer rebalances
    * Released: 2.4.0
    * Last modified: Sep 09 2019
* KIP-429: Kafka Consumer Incremental Rebalance Protocol
    * Released: 2.4.0
    * Last modified: Mar 16 2021

P113

One of the key aspect of this protocol is that, as a developer, we can embed our own protocol to customize how partitions are assigned to the group members.

In this post, we will see which strategies can be configured for Kafka Client Consumer and how to write a custom `PartitionAssignor` implementing a failover strategy.

When creating a new Kafka consumer, we can configure the strategy that will be used to assign the partitions amongst the consumer instances.

The assignment strategy is configurable through the property `partition.assignment.strategy`

A strategy is simply the fully qualified name of a class implementing the interface `PartitionAssignor`.

Kafka Clients provides three built-in strategies : **Range**, **RoundRobin** and **StickyAssignor**.

P114

The following code snippet illustrates how to specify a partition assignor :

```
Properties props = new Properties();
...
props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG, StickyAssignor.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

//...
```

P115

All consumers which belong to the same group must have one common strategy declared. If a consumer attempts to join a group with an assignment configuration inconsistent with other group members, you will end up with this exception :

```
org.apache.kafka.common.errors.InconsistentGroupProtocolException: The group member's supported protocols are incompatible with those of existing members or first group member tried to join with empty protocol type or empty protocol list.
```

This property accepts a comma-separated list of strategies. For example, it allows you to update a group of consumers by specifying a new strategy while temporarily keeping the previous one. Part of the Rebalance Protocol the broker coordinator will choose the protocol which is supported by all members.

P116

The *RangeAssignor* is the default strategy. The aims of this strategy is to co-localized partitions of several topics. This is useful, for example, to join records from two topics which have the same number of partitions and the same key-partitioning logic.

For doing this, the strategy will first put all consumers in lexicographic order using the *member_id* assigned by the broker coordinator. Then, it will put available topic-partitions in numeric order. Finally, for each topic, the partitions are assigned starting from the first consumer .
As you can seen, partitions 0 from topics A and B are assigned to the same consumer.

In the example, at most two consumers are used because we have maximum of two partitions per topic . If you plan to consume from multiple input topics and you are not performing an operation requiring to co-localized partitions you should definitely not use the default strategy.

P117

The `RoundRobinAssignor` can be used to distribute available partitions evenly across all members. As previously, the assignor will put partitions and consumers in lexicographic order before assigning each partitions.

Even if RoundRobin provides the advantage of maximizing the number of consumers used, it has one major drawback. Indeed, it does not attempt to reduce partition movements when the number of consumers changes (i.e. when a rebalance occurs).

P118

To illustrate this behaviour, let's remove the consumer 2 from the group. In this scenario, topic-partition B-1 is revoked from C1 to be re-assigned to C3. Conversely topic-partition B-0 is revoked from C3 to be re-assigned to C1.
For example, if a consumer initializes internal caches, opens resources or connections during partition assignment, this unnecessary partition movement can have an impact on consumer performance.

P119

The `StickyAssignor` is pretty similar to the RoundRobin except that it will try to minimize partition movements between two assignments, all while ensuring a uniform distribution.

Using the previous example, if consumer C2 leaves the group then only partition A-1 assignment changes to C3.

P120

Implementing a Custom Strategy

The PartitionAssignor interface

The *PartitionAssignor* is not so much complicated and only contains four main methods.

```
public interface PartitionAssignor {

Subscription subscription(Set<String> topics);

Map<String, Assignment> assign(
Cluster metadata,
Map<String, Subscription> subscriptions);

void onAssignment(Assignment assignment);

String name();
}
```

First, the `subscription()` method is invoked on all consumers, which are responsible to create the *Subscription* that will be sent to the broker coordinator. A *Subscription* contains the set of topics that consumer subscribes to and, optionally, some user-data that may be used by the assignment algorithm.

Then, part of the Rebalance Protocol the consumer group leader will receives the subscription from all consumers and will be responsible to perform the partition assignment through the method `assign()` .

Next, all consumers will receive their assignment from the leader and the `onAssignment()` method will be invoked on each. This method can be used by consumers to maintain internal state.

Finally, a `PartitionAssignor` must be assigned to a unique name returned by the method `name()` (e.g. “range” or “roundrobin” or “sticky”).

P121

Failover strategy

With default assignors all consumers in a group can be assigned to partitions. We can compare this strategy to an active/active model which means that all instances will potentially fetch messages at the same time. But, for some production scenarios, it may be necessary to perform an active/passive consumption. Hence, I propose to you to implement

a *FailoverAssignor* which is actually a strategy that can be found in some other messaging solutions.

The basic idea behind Failover strategy is that multiple consumers can join a same group. However, all partitions are assigned to a single consumer at a time. If that consumer fails or is stopped then partitions are all assigned to the next available consumer. Usually, partitions are assigned to the first consumer but for our example we will attach a priority to each of our instance. Thus, the instance with the highest priority will be preferred over others.

Let's illustrate this strategy. In the example below, C1 has the highest priority, so all partitions are assigned to it

P122

If the consumer fails, then all partitions are assigned to the next consumer (i.e C2).

Implementation link:

https://medium.com/streamthoughts/understanding-kafka-partition-assignment-strategies-and-how-to-write-your-own-custom-assignor-ebeda1fc06f3

P124

Four partitions distributed across three brokers

With Kafka the unit of replication is the partition. Each topic has one or more partitions and each partition has a leader and zero or more followers. When you create a topic you specify the number of partitions and the replication factor. A replication factor of three is common, this equates to one leader and two followers. Both leaders and followers can be referred to as replicas.

P125

Producer, Consumer And Leader

All reads and writes on a partition go to the leader. Followers periodically send fetch requests to the leader to get the latest messages. Consumers do not consume from followers, the followers only exist for redundancy and fail-over.

P126

Partition Fail-Over - Broker 3 dies

When a broker dies, it will likely be the host of multiple partition leaders. For each partition that has lost a leader, a follower on a remaining node will be promoted to leader. In fact this isn't always the case as it depends on whether there are followers that are "in-sync" with the leader, and if not, whether fail-over to an out-of-sync replica is allowed. But for now let's keep it simple.
When Broker 3 dies, a new leader on broker 2 is elected for partition 2.

P127

Partition Fail-Over - Broker 1 dies too

Then when broker 1 dies, partition 1 loses its leader and it fails-over to broker 2.

P128

Leaders remain on broker 2

When Broker 1 comes back online it adds four followers giving some redundancy to each partition. But the leaders remain concentrated on broker 2.

P129

Unbalanced leaders after recovery of broker 1 and 3

When broker 3 comes back we are back to three replicas per partition. But still all leaders are hosted on broker 2.

P130

Replica leaders rebalanced

Kafka has the concept of preferred replica leaders. When Kafka creates the partitions of a topic, it tries to distribute the leaders of each partition evenly across the nodes and marks those first leaders as the preferred leaders. Over time, due to server restarts, server failures and network partitions, the leaders might end up on different nodes to the preferred replica. Just like in the more extreme case above.To fix that Kafka offers two options:

1. The topic configuration **auto.leader.rebalance.enable=true** which allows the controller node to reassign leadership back to the preferred replica leaders and thereby restore even distribution. (**leader.imbalance.check.interval.seconds** takes to apply)
2. An administrator can use the kafka-preferred-replica-election.sh script to perform it manually.
    - Above tool was deprecated, use ./kafka-leader-election.sh script to perform it manually

P132

Controller Broker

A distributed system must be coordinated. If some event happens, the nodes in the system must react in an organized way. In the end, somebody needs to decide on how the cluster reacts and instruct the brokers to do something.

That somebody is called a ***Controller***. A controller is not too complex — it is a normal broker that simply has additional responsibility. That means it still leads partitions, has writes/reads going through it and replicates data.
The most important part of that additional responsibility is keeping track of nodes in the cluster and appropriately handling nodes that leave, join or fail. This includes rebalancing partitions and assigning new partition leaders.
There is always exactly one controller broker in a Kafka cluster.

P133

Handle a node leaving the cluster

When a node leaves the Kafka cluster, either due to a failure or intentional shutdown, the partitions that it was a leader for will become unavailable (remember that clients only read from/write to partition leaders). Thus, to minimize downtime, it is important to find substitute leaders as quickly as possible.

A Controller is the broker that reacts to the event of another broker failing. It gets notified from a ZooKeeper Watch. A ZooKeeper Watch is basically a subscription to some data in ZooKeeper. When said data changes, ZooKeeper will notify everybody who is subscribed to it. ZooKeeper watches are crucial to Kafka — they serve as input for the Controller.

The tracked data in question here is the set of brokers in the cluster.

As shown below, Broker 2's id is deleted from the list due to the expiry of the faulty broker's ZooKeeper Session *(Every Kafka node heartbeats to ZooKeeper and this keeps its session alive. Once it stops heartbeating, the session expires)*.

The controller gets notified of this and acts upon it. It decides which nodes should become the new leaders for the affected partitions. It then informs every associated broker that it should either become a leader or start replicating from the new leader via a `LeaderAndIsr` request.

P134

Split Brain

Imagine a controller broker dies. The Kafka cluster must find a substitute, otherwise it can quickly deteriorate in health when there is nobody to maintain it.

There is the problem that you cannot truly know whether a broker has stopped for good or has experienced an intermittent failure. Nevertheless, the cluster has to move on and pick a

new controller. We can now find ourselves having a so-called **zombie controller**. A zombie controller can be defined as a controller node which had been deemed dead by the cluster and has come back online. Another broker has taken its place but the zombie controller might not know that yet.

This can easily happen. For example, if a nasty intermittent network partition happens or a controller has a long enough stop-the-world GC pause — the cluster will think it has died and pick a new controller. In the GC scenario, nothing has changed through the eyes of the original controller. The broker does not even know it was paused, much less that the cluster moved on without it. Because of that, it will continue acting as if it is the current controller. This is a common scenario in distributed systems and is called split-brain.

Let's go through an example. Imagine the active controller really does go into a long stop-the-world GC pause. Its ZooKeeper session expires and `/controller` znode it registered is now deleted. Every other broker in the cluster is notified of this as they placed ZooKeeper Watches on it.

P135

To fix the controller-less cluster, every broker now tries to become the new controller itself. Let's say Broker 2 won the race and became the new controller by creating the `/controller` znode first.

Every broker receives a notification that this znode was created and now knows who the latest leader is — Broker 2. Every broker except Broker 3, which is still in a GC pause. It is possible that this notification does not reach it for one reason or another (e.g OS has too many accepted connections awaiting processing and drops it). In the end, the information about the leadership change does not reach Broker 3.

P136

Broker 3's garbage collection pause will eventually finish and it will wake up still thinking it is in charge. Remember, nothing has changed through its eyes.

You now have two controllers which will be giving out potentially conflicting commands out in parallel. This is something you obviously do not want to happen in your cluster. If not handled, it can result in major inconsistencies.

If Broker 2 (new controller node) receives a request from Broker 3, how will it know whether Broker 3 is the newest controller or not? For all Broker 2 knows, the same GC pause might have happened to it too!

There needs to be a way to distinguish who the real, current controller of the cluster is.

P137

There is such a way! It is done through the use of an ***epoch number*** (also called a fencing token). An epoch number is simply a monotonically increasing number — if the old leader had an epoch number of 1, the new one will have 2. Brokers can now easily differentiate the real controller by simply trusting the controller with the highest number. The controller with the highest number is surely the latest one, since the epoch number is always-increasing. This epoch number is stored in ZooKeeper (leveraging its consistency guarantees).

Here, Broker 1 stores the latest `controllerEpoch` it has seen and ignores all requests from controllers with a previous epoch number.

Other responsibilities

The controller does other, more boring things too.

- Create new partitions
- Create a new topic
- Delete a topic

Previously, these commands could only be done in a whacky way — a bash script which directly modified ZooKeeper and waited for the controller broker to react to the changes. Since version 0.11 and 1.0, these commands have been changed to be direct requests to the

controller broker itself. They are now easily callable by any user app through the AdminClient Api which sends a request to the controller.

P139

One of the main problems we are encountering these days are the amount of disk space used by Apache Kafka topics. Our messages are serialized as JSON. As you know in JSON, each field of the data model is stored in the JSON string, which causes a lot of duplicated field names to be stored in the Kafka topic. To reduce disk space usage, we have several options and I want to share some of them here.

Why do I need to reduce message size?

Although you may argue that storage is cheap today, there are several situations that you want to reduce disk space and bandwidth:

- It is not strange to store all of your data inside the Kafka. New York Times, for example, uses Kafka as a source of truth! It is possible to keep the entire topic records forever by disabling retention in Kafka. With correct compression, you can reduce the

amount of disk space usage significantly. If you don't reduce message size, you might run out of disk space several times faster than you think!

- If you can reduce the size of messages when you send them to the broker, you use smaller bandwidth and therefore you can increase total numbers of messages that can be sent to the brokers.
- Cloud platforms offer pricing that is calculated by the amount of data that is written to or read from the Kafka cluster and the amount of data that is stored in the cluster! So reducing these factors can significantly reduce your application costs.

Before going to show you how we can reduce the message size, I want to describe Kafka's message format.

P140

Message format

The message format of Kafka is already described in its documentation, so I discuss it briefly.

For this article it is enough to define record, record batch and record batch overhead:

- Record: each record is our familiar key and value pair (and some small additional data).
- Record batch: each produce request (for a topic partition) that is sent to a Kafka broker is wrapped in a batch. Each batch contains one to several records and it contains a record batch overhead section.
- Record batch overhead: each produced batch contains metadata about the records such as message version (magic byte), records count, compression algorithm, transaction and so on. This metadata is stored in the overhead section of the batch. Record batch overhead is **61 bytes.**

Record batch overhead is constant and we cannot reduce the size of it. But we can optimize the size of the record batch in 3 ways: **lingering**, **compression** and **using schemas** for our records values and keys.

P141

Lingering

How public transportation helps us to reduce traffic?

As you see above if each person uses his/her car to travel in the city, we might face heavy traffic. But on the other hand, using buses to transfer passengers to their destination can decrease traffic significantly. This will be achieved because we move the same people by using a smaller space (buses).

P142

Lingering in Kafka helps us to do the same. Instead of sending a batch with a single record, we can wait a bit more and gather more records in produce time, and send them as a batch: You can enable it by setting `linger.ms` config to the number of milliseconds that your producer must wait for gathering our 100 records.

You can enable it by setting `linger.ms` config to the number of milliseconds that your producer must wait for gathering our 100 records.

P143

Compression

Compression helps us to reduce the amount of data we want to store on the disk or send over the network at the cost of more CPU usage. This is a trade-off between more IO or more CPU usage. Most web applications today spend their time waiting for IO (network, database query and ..) and CPU usage might be small. Hence it makes sense to use CPU to reduce IO in this situation.

In Kafka producer, before sending the record batch to the broker, we can use compression to reduce the size of the batch. The producer compresses only the records inside the batch and does not compress the batch overhead section. When compression enabled, a flag bit is set for compression type in the batch overhead. This flag will be used during decompression in the consumer.

Compression algorithms work well when there is more duplication in the data. The more records you have in the batch, the higher the compression ratio you can expect. That is why the producer compresses all the records in the batch together (instead of compressing each

record separately). For example, in JSON it is more likely to have the same field names in the adjacent records.

Kafka supports various compression algorithms. If you want to have a higher compression ratio you can use gzip in the cost of more CPU usage. But if you want less CPU usage and faster algorithm, you can choose Snappy. You can set the compression algorithm using `compression.type` config in the producer.

P144

Kafka Clients

In this new version of Kafka, the biggest change by far is the Support for ZStandard Compression (KIP-110).

Support for Zstandard Compression (KIP-110)

This algorithm was created by Facebook in September 2016 and it has been for 2 years in the works. The compression ratio is as good as **gzip** on the first pass. But, if we now look at the compression and decompression speed, which means how many megabytes can be encrypted or decrypted per second, we have 5 times the performance of gzip.

	Ratio	Comp Speed MB/s	Decomp Speed MB/s
zstd cli	3.14	100.1	459.40
gzip cli	3.11	19.66	125.55

In Kafka, according to the KIP, the compression works great, as we have an 4.28 compression ratio. Shopify is an example of a production environment using ZStandard and they noted a massive decrease in CPU usage. Overall, if you use ZStandard you should have more throughput for a fraction of the cost.

Here is how to use ZStandard compression:

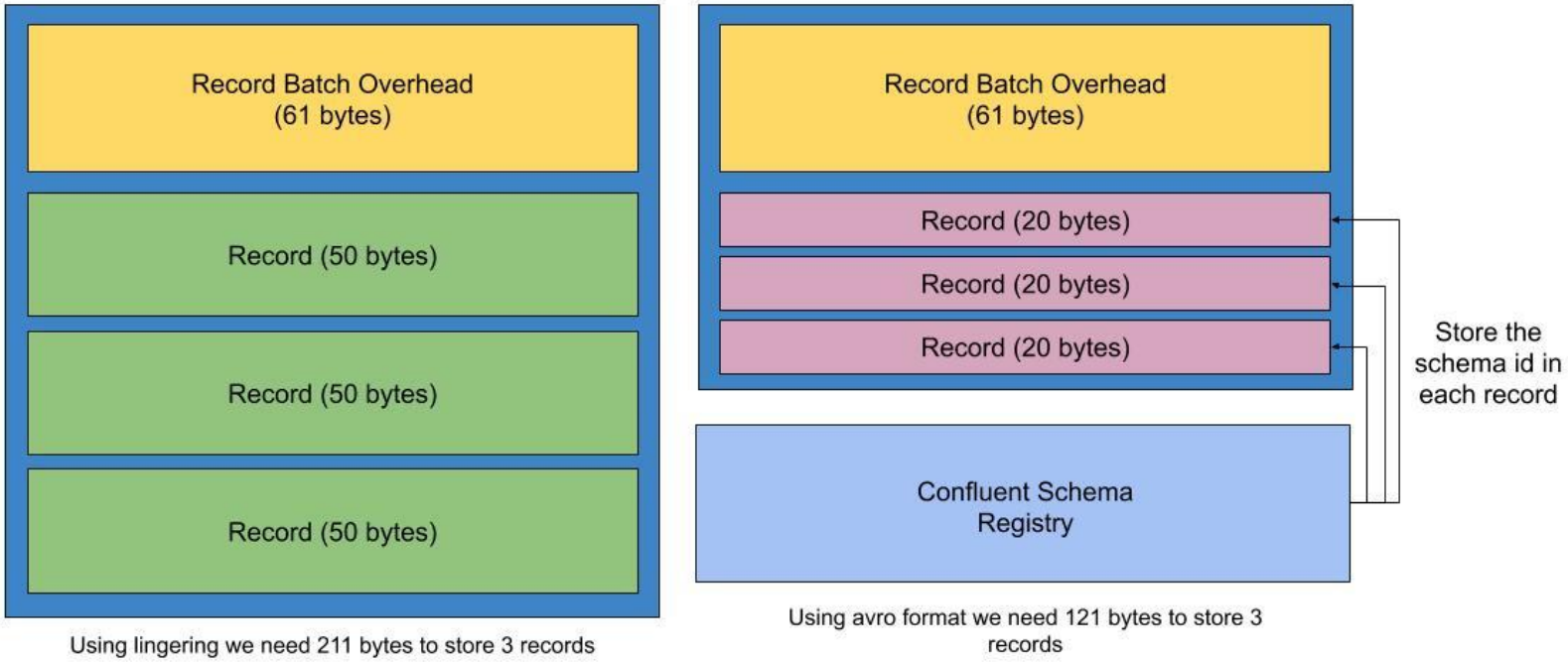
- **Producers(2.1)**: `compression.type=zstd`
- **Consumers (<2.1)**: UNSUPPORTED_COMPRESSION_TYPE (error)
- **Consumers(>=2.1)**: It should be working out of the box

Also, to use zstd compression, **you also have to update the brokers to 2.1**. To summarize, if you want to use zstd, the first thing you have to do is update your consumers, then your brokers and finally your producers.

P145

Schema

Using schema to store Kafka records, has several benefits and one of them is the reduction of record size. Because in JSON you have to store the name of each field with your data in the record, the total size of the record increases significantly. But if you choose to store your data in Avro format, you can store schema once and produce records based on that schema many times. This way you reduce the size of the record by removing the schema from the record. You just need to store the schema of the record value (or key) in Confluent Schema Registry and keep the schema id in the record.



Storing schema in a separate place makes sense because the schema is the same for most of the records and if you have a newer version of schema you can store the new one in Confluent Schema Registry.

P146

Conclusion

We compared several approaches that can be used in Apache Kafka to reduce disk space usage. As you see each one has its trade-off:

- Lingering needs you wait a bit more to gather more records.
- Compression uses more CPU usage but will reduce the amount of IO.
- Using Avro impose a dependency on clients (consumer and producer) to have another data store (Confluent Schema Registry) to keep the schema of the records.

In some cases, these trade-offs should be acceptable and reduce resource usage.

P147

A message sent to a Kafka cluster is appended to the end of one of the logs. The message remains in the topic for a configurable period of time or until a configurable size is reached until the specified retention for the topic is exceeded. The message stays in the log even if the message has been consumed.

If the log retention is set to five days, then the published message is available for consumption five days after it is published. After that time, the message will be de discarded to free up space. The performance of Kafka is not affected by the data size of messages, so retaining lots of data is not a problem.

Apache Kafka provides two types of Retention Policies. Time base and size base

P148

Time Based Retention:

Once the configured retention time has been reached for Segment, it is marked for deletion or compaction depending on configured cleanup policy. Default retention period for Segments is 7 days.

Config Name	Description	Default Value	Valid Values
log.retention.ms	The number of milliseconds to keep a log file before deleting it (in milliseconds), If not set, the value in log.retention.minutes is used	null	any long value
log.retention.minutes	The number of minutes to keep a log file before deleting it (in minutes), secondary to log.retention.ms property. If not set, the value in log.retention.hours is used	null	any long value
log.retention.hours	The number of hours to keep a log file before deleting it (in hours), tertiary to log.retention.ms property	168	any long value

P149

Size Based Retention:

In this policy, we configure the maximum size of a Log data structure for a Topic partition. Once Log size reaches this size, it starts removing Segments from its end. This policy is not popular as this does not provide good visibility about message expiry. However it can come handy in a scenario where we need to control the size of a Log due to limited disk space.

Config Name	Description	Default Value	Valid Values
log.retention.bytes	The maximum size of the log before deleting it	-1	any long value

So till now we understood what are retention policies, once retention period is reached, clean policies comes into picture. Lets understand Cleanup Policies now.

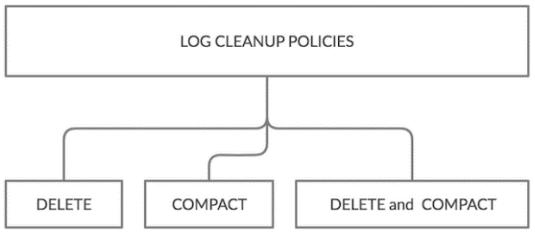
P150

Log Cleanup Policies! What are these?

In Kafka, unlike other messaging systems, the messages on a topic are not immediately removed after they are consumed. Instead, the configuration of each topic determines how much space the topic is permitted and how it is managed.

Concept of making data expire is called as Cleanup. Its a Topic level configuration. It is important to restrict log segment to continue grow in size.

Types of Cleanup Policies:



Delete Policy:

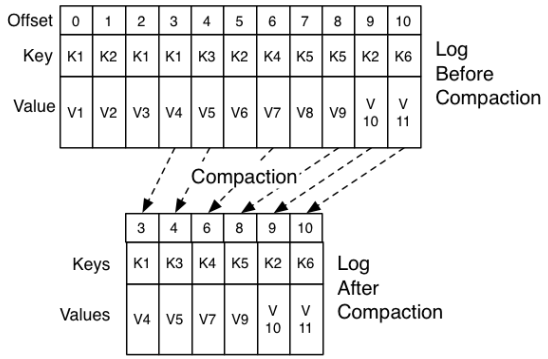
This is default cleanup policy. This will discard old segments when their retention time or size limit has been reached.

P151

Compact Policy:

This will enable **Log Compaction** on a topic. The idea is to selectively remove records for each partition where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

Lets see this first, high level digram of a immutable log stream for a single topic demonstrating compaction.



As we can see here it does cleanup of values V1 for key K1 and keeps latest copy with value as V4.

Here are few important configurations for Log Compaction

Config Name	Description	Default Value	Valid Values
log.cleaner.min.compaction.lag.ms'	The minimum time a message will remain uncompacted in the log. Only applicable for logs that are being compacted.	0	any long value
log.cleaner.max.compaction.lag.ms'	The maximum time a message will remain ineligible for compaction in the log. Only applicable for logs that are being compacted.	9223372036854775807	any long value

Delete and Compact Both:

We can specify both `delete` and `compact` values for the `cleanup.policy` configuration at the same time. In this case, the log is compacted, but the cleanup process also follows the retention time or size limit settings.

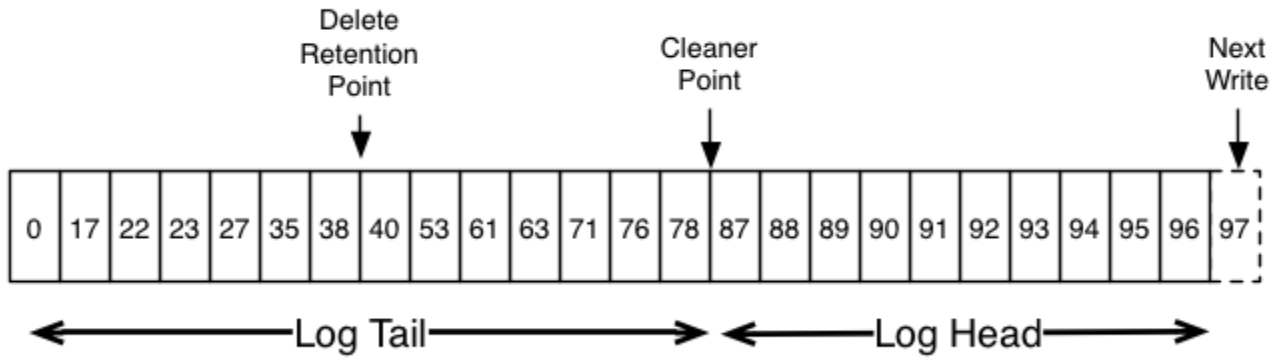
When both methods are enabled, capacity planning is simpler than when you only have compaction set for a topic. However, some use cases for log compaction depend on messages not being deleted by log cleanup, so consider whether using both is right for your scenario.

P152

What is Log Cleaner?

Log cleaner does Log compaction. Log cleaner is a pool of background compaction threads.

How each compaction thread works?



1. It chooses the log that has the highest ratio of log head to log tail
2. It creates a succinct summary of the last offset for each key in the head of the log
3. It recopies the log from beginning to end removing keys which have a later occurrence in the log. New, clean segments are swapped into the log immediately so the additional disk space required is just one additional log segment (not a fully copy of the log).
4. The summary of the log head is essentially just a space-compact hash table. It uses exactly 24 bytes per entry. As a result with 8GB of cleaner buffer one cleaner iteration can clean around 366GB of log head (assuming 1k messages).

How to enable Log Cleaner and other configurations

Config Name	Description	Default Value	Valid Values
log.cleaner.enable	Enable the log cleaner process to run on the server. Should be enabled if using any topics with a cleanup.policy=compact including the internal offsets topic. If disabled those topics will not be compacted and continually grow in size.	true	true or false
log.cleaner.threads	The number of background threads to use for log cleaning	1	[0,...]
log.cleaner.io.max.bytes.per.second	The log cleaner will be throttled so that the sum of its read and write i/o will be less than this value on average	1.7976931348623157E308	any double value
log.cleaner.dedupe.buffer.size	The total memory used for log deduplication across all cleaner threads	134217728	any long value
log.cleaner.io.buffer.size	The total memory used for log cleaner I/O buffers across all cleaner threads	524288	[0,...]
log.cleaner.io.buffer.load.factor	Log cleaner dedupe buffer load factor. The percentage full the dedupe buffer can become. A higher value will allow more log to be cleaned at once but will lead to more hash	0.9	
log.cleaner.backoff.ms'	The amount of time to sleep when there are no logs to clean	15000	[0,...]

P153

Few FAQs about Log Cleaner?

1. **Does Log Cleaner impacts Read performance?**

No. Cleaning does not block reads and can be throttled to use no more than a configurable amount of I/O throughput to avoid impacting producers and consumers.

2. Does offset of message gets changed after compaction?

No. The offset for a message never changes. It is the permanent identifier for a position in the log.

3. Does Log Cleaner also deletes Tombstone messages?

Yes. A message with a key and null payload is considered as Tombstone message. Log cleaner also deletes these tombstones.

4. Does order of messages gets changed after compaction?

No. Ordering of messages is always maintained. Compaction will never re-order messages, just remove some.

P155

Apache Kafka and the need for security

Apache Kafka is an internal middle layer enabling your back-end systems to share real-time data feeds with each other through Kafka topics. **With a standard Kafka setup, any user or application can write any messages to any topic, as well as read data from any topics**. As your company moves towards a shared tenancy model where multiple teams and applications use the same Kafka Cluster, or your Kafka Cluster starts on boarding some critical and confidential information, you need to implement security.

P156

Problems Security is solving

Kafka Security has three components:

- **Encryption of data in-flight using SSL / TLS:** This allows your data to be encrypted between your producers and Kafka and your consumers and Kafka. This is a very common pattern everyone has used when going on the web. That's the "S" of HTTPS (that beautiful green lock you see everywhere on the web).
- **Authentication using SSL or SASL:** This allows your producers and your consumers to authenticate to your Kafka cluster, which verifies their identity. It's also a secure way to enable your clients to endorse an identity. Why would you want that? Well, for authorization!
- **Authorization using ACLs:** Once your clients are authenticated, your Kafka brokers can run them against access control lists (ACL) to determine whether or not a particular client would be authorised to write or read to some topic.

P157

Encryption solves the problem of the man in the middle (MITM) attack. That's because your packets, while being routed to your Kafka cluster, travel your network and hop from machines to machines. If your data is PLAINTEXT (by default in Kafka), any of these routers could read the content of the data you're sending:

Now with Encryption enabled and carefully setup SSL certificates, your data is now encrypted and securely transmitted over the network. With SSL, only the first and the final machine possess the ability to decrypt the packet being sent.

This encryption comes at a cost: CPU is now leveraged for both the Kafka Clients and the Kafka Brokers in order to encrypt and decrypt packets. SSL Security comes at the cost of performance, but it's low to negligible. Using Java 9 vs Java 8 with Kafka 1.0 or greater, the performance cost is decreased further by a substantial amount!

P158

There are two ways to authenticate your Kafka clients to your brokers: SSL and SASL. Let's go over both

SSL Authentication

SSL Auth is basically leveraging a capability from SSL called two ways authentication. The idea is to also issue certificates to your clients, signed by a certificate authority, which will allow your Kafka brokers to verify the identity of the clients.

This is the most common setup I've seen when you are leveraging a managed Kafka clusters from a provider such as Heroku, Confluent Cloud or CloudKarafka.

P159

Once your Kafka clients are authenticated, Kafka needs to be able to decide what they can and cannot do. This is where Authorization comes in, controlled by Access Control Lists (ACL). **ACL are what you expect them to be: User A can('t) do Operation B on Resource C from Host D**. Please note that currently with the packaged `SimpleAclAuthorizer` coming with Kafka, ACL are not implemented to have Groups rules or Regex-based rules. Therefore, each security rule has to be written in full (with the exception of the * wildcard).

ACL are great because they can help you prevent disasters. For example, you may have a topic that needs to be writeable from only a subset of clients or hosts. You want to prevent your average user from writing anything to these topics, hence preventing any data corruption or deserialization errors. ACLs are also great if you have some sensitive data and you need to prove to regulators that only certain applications or users can access that data.