

# Apache Kafka® Administration by Confluent

Student Handbook

Version 6.0.0-v1.0.0



CONFLUENT

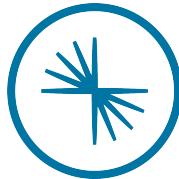
tejaswin.renugunta@wdgens.com

# Table of Contents

<b>01: Introduction</b> .....	<b>1</b>
<b>02 Fundamentals of Apache Kafka</b> .....	<b>9</b>
<b>03: Providing Durability</b> .....	<b>41</b>
<b>04: Managing a Kafka Cluster</b> .....	<b>125</b>
<b>05: Optimizing Kafka's Performance</b> .....	<b>198</b>
<b>06: Kafka Security</b> .....	<b>271</b>
<b>07 Data Pipelines with Kafka Connect</b> .....	<b>344</b>
<b>08: Kafka in Production</b> .....	<b>404</b>
<b>09: Conclusion</b> .....	<b>471</b>
<b>Appendix A: Important JMX Metrics</b> .....	<b>478</b>

tejaswin.renugunta@walgreens.com

# 01: Introduction



CONFLUENT

tejaswin.renugunta@walgreens.com

# Copyright & Trademarks

Copyright © Confluent, Inc. 2014-2020. [Privacy Policy](#) | [Terms & Conditions](#).

Apache, Apache Kafka, Kafka and the Kafka logo are trademarks of the

[Apache Software Foundation](#)

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction ... ←
2. Fundamentals of Apache Kafka
3. Providing Durability
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance
6. Kafka Security
7. Data Pipelines with Kafka Connect
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives



After this course you will be able to:

- Describe how Kafka Brokers, Producers & Consumers work
- Describe how replication works within the cluster
- List hardware and runtime configuration options
- Monitor and administer your Kafka cluster
- Secure your Kafka cluster
- Integrate Kafka with external systems using Kafka Connect
- Design a Kafka cluster for high availability & fault tolerance

Throughout the course, Hands-On Exercises will reinforce the topics being discussed

tejaswin.renugunta@walgreens.com

# Prerequisite

This course requires a working knowledge of the Kafka architecture.

New to Kafka? Need a refresher?

Sign up for free "Confluent Fundamentals for Apache Kafka" course at  
<https://confluent.io/training>

---

Attendees should have a working knowledge of the Kafka architecture, either from prior experience or the recommended prerequisite course Confluent Fundamentals for Apache Kafka®.

This free course is available at <http://confluent.io/training> for students who need to catch up.

tejaswin.renugunta@walgreens.com

# Other Confluent Training Courses

- Confluent Developer Skills for Building Apache Kafka®
- Confluent Stream Processing using Apache Kafka® Streams & ksqlDB
- Confluent Advanced Skills for Optimizing Apache Kafka®



For more details, see <https://confluent.io/training>

---

- **Confluent Developer Skills for Building Apache Kafka®** covers:
  - Write Producers and Consumers to send data to & read data from Kafka
  - Integrate Kafka with external systems using Kafka Connect
  - Write streaming applications with Kafka Streams & ksqlDB
  - Integrate a Kafka client application with Confluent Cloud
- **Confluent Stream Processing using Apache Kafka® Streams & ksqlDB** covers:
  - Identify common patterns and use cases for real-time stream processing
  - Describe the high level architecture of Kafka Streams
  - Use Kafka Streams and ksqlDB to filter, transform, enrich, aggregate, and join data streams in real-time
  - Test, secure, deploy, and monitor Kafka Streams applications and ksqlDB queries
- **Confluent Advanced Skills for Optimizing Apache Kafka®**
  - Formulate the Apache Kafka® Confluent Platform specific needs of your company
  - Monitor all essential aspects of your Confluent Platform
  - Tune the Confluent Platform according to your specific needs
  - Provide first level production support for your Confluent Platform

# Class Logistics



- Start and end times
- Can I come in early/stay late?
- Breaks
- Lunch
- Restrooms
- Wi-Fi and other information
- Emergency procedures



No recordings please (audio, video)

tejaswin.renugunta@walgreens.com

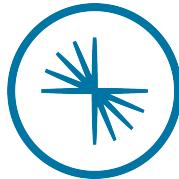
# Introductions



- About you:
  - Your Name, Company and Role
  - Your Experience with Kafka
  - Other Messaging or Big Data Systems, you use
  - OS, Programming Languages
  - Your Course Expectations
- About your instructor

tejaswin.renugunta@walgreens.com

# 02 Fundamentals of Apache Kafka



CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Fundamentals of Apache Kafka ... ←
3. Providing Durability
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance
6. Kafka Security
7. Data Pipelines with Kafka Connect
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

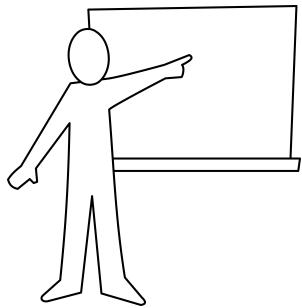


After this module you will be able to:

- explain the value of a **Distributed Event Streaming Platform**
- explain how the “log” abstraction enables a distributed event streaming platform
- explain the basic concepts of:
  - Brokers, Topics, Partitions, and Segments
  - Records (a.k.a. Messages, Events)
  - Producers, Consumers, and Serialization
  - Replication

tejaswin.renugunta@walgreens.com

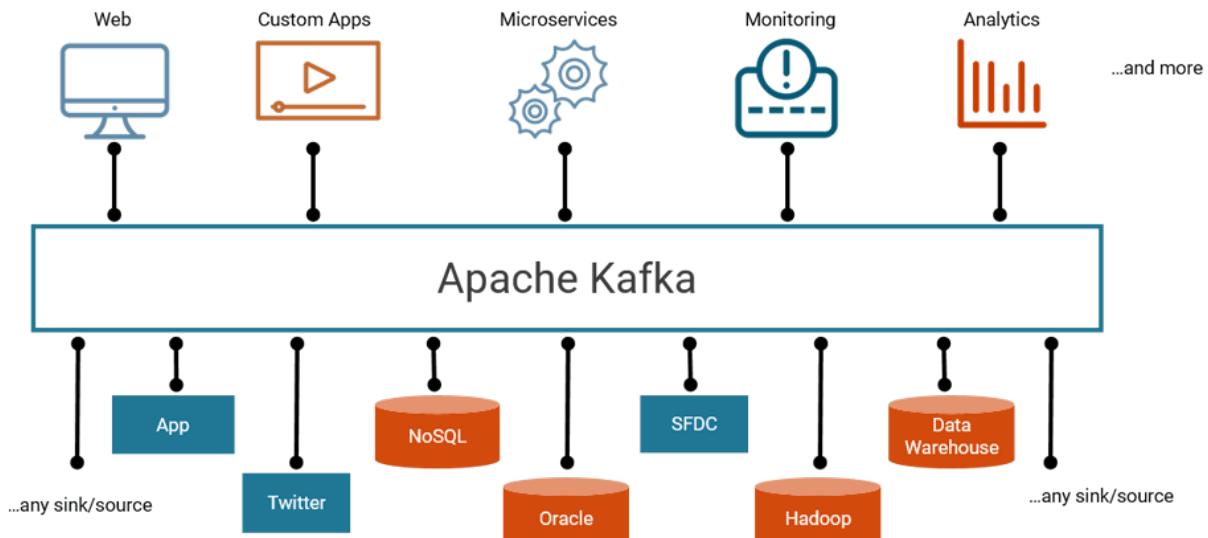
# Module Map



- Event Streaming Platform ... ←
- Event Streaming Architecture
- Replication
- Hands-on Lab: **Introduction**
- Hands-on Lab: **Using Kafka's Command-Line Tools**
- Hands-on Lab: **Consuming from Multiple Partitions**

tejaswin.renugunta@walgreens.com

# The Streaming Platform



Apache Kafka is the foundation of what we call a **Distributed Streaming Platform**. Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, high throughput, low latency, and runs in production at thousands of companies.

A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur, in real-time.

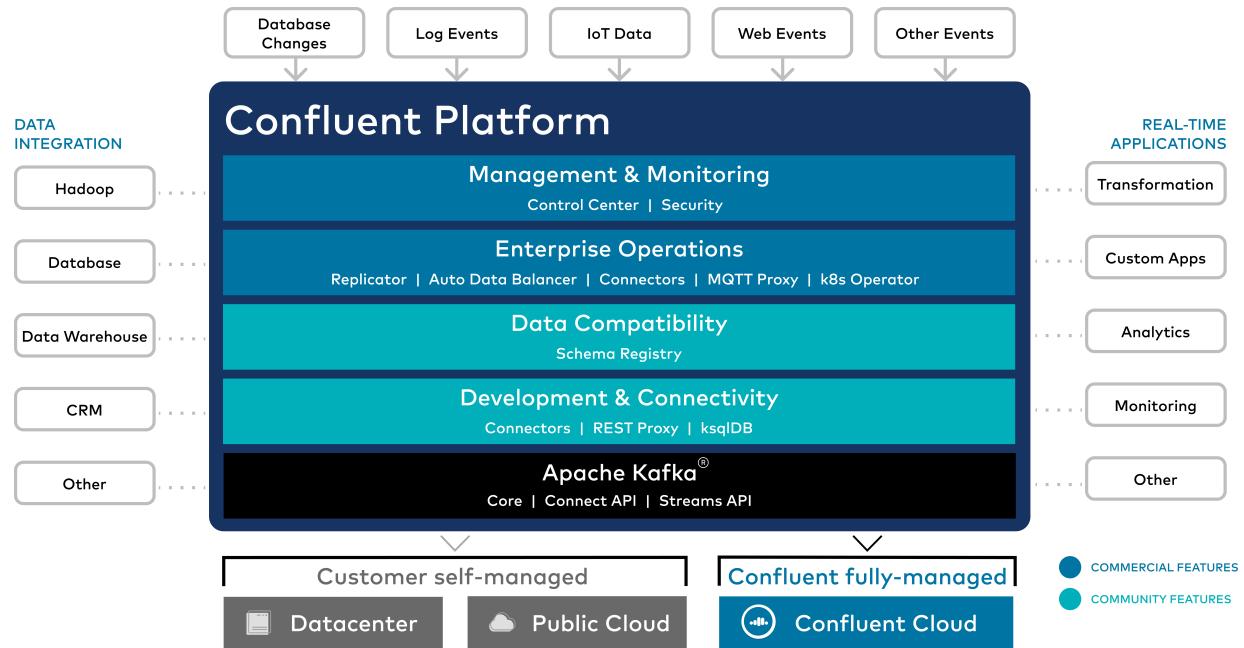
Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data



SFDC stands for Salesforce Dot Com

# Confluent Platform, Built on Apache Kafka

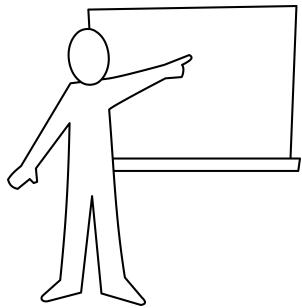


The purpose of this slide is to clarify the open source and commercial components of Confluent Platform. Here is a sample of components and their licenses:

Component	License	Description
Core Apache Kafka	Apache 2.0	Scalable, fault tolerant, high throughput, low latency distributed event storage, as well as the Producer and Consumer client APIs used to interact with event logs.
Kafka Connect API	Apache 2.0	A framework for creating connectors to get data in and out of Kafka.
Kafka Streams API	Apache 2.0	The Kafka Streams API allows developers to create scalable, fault tolerant, near real-time stream processing applications.
Confluent ksqlDB	Confluent Community	ksqlDB allows developers to create powerful Kafka Streams applications using a simple SQL-like syntax.
Schema Registry	Confluent Community	Schema Registry coordinates schema serialization and deserialization and allows schemas to evolve more easily.
Confluent Security Plugins	Confluent Enterprise	Various plugins allow for advanced security features: Role Based Access Control, LDAP Authorizer, Confluent REST Proxy principal propagation, etc..
Confluent Operator	Confluent Enterprise	A Kubernetes Operator that makes it easy to deploy and manage the Confluent Platform in Kubernetes.

We will learn about these and other components throughout the course. For more information about Confluent Platform components, see <https://docs.confluent.io/current/platform.html>.

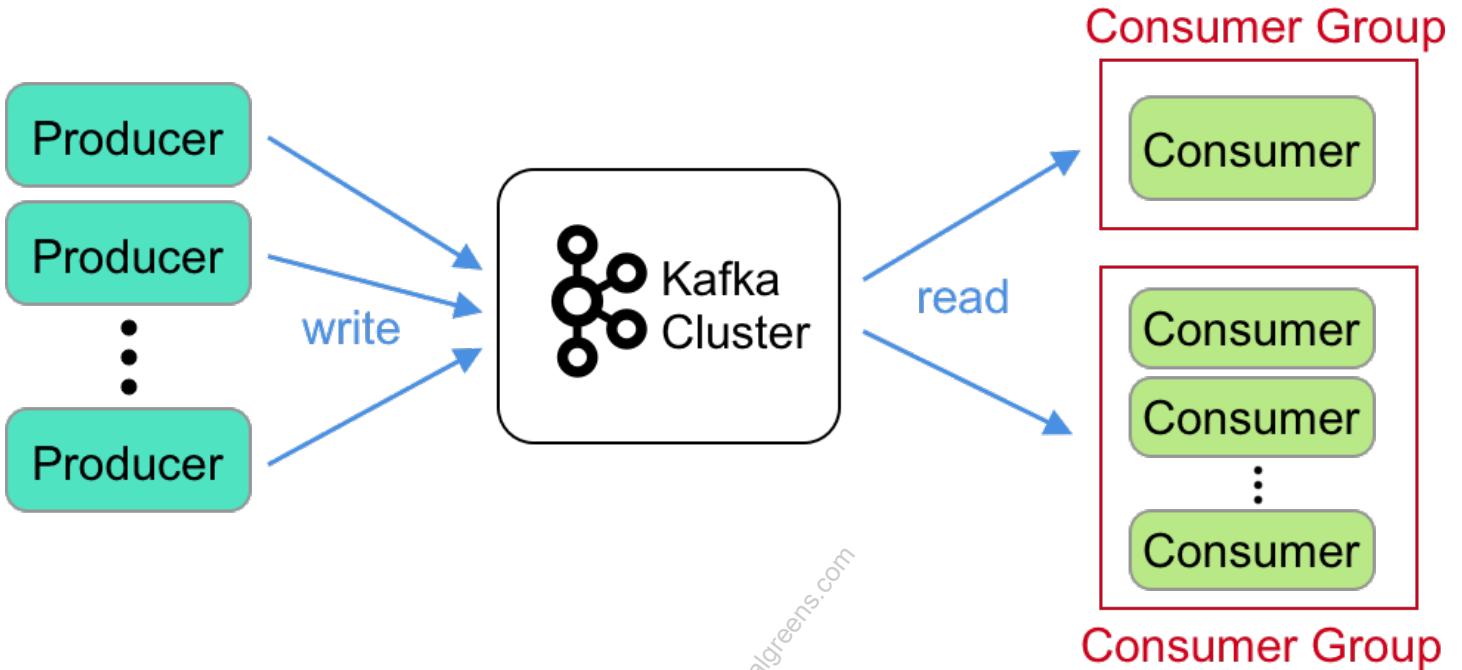
# Module Map



- Event Streaming Platform
- Event Streaming Architecture ... ↵
- Replication
- ⚡ Hands-on Lab: **Introduction**
- ⚡ Hands-on Lab: **Using Kafka's Command-Line Tools**
- ⚡ Hands-on Lab: **Consuming from Multiple Partitions**

tejaswin.renugunta@walgreens.com

# Kafka Clients - Producers & Consumers



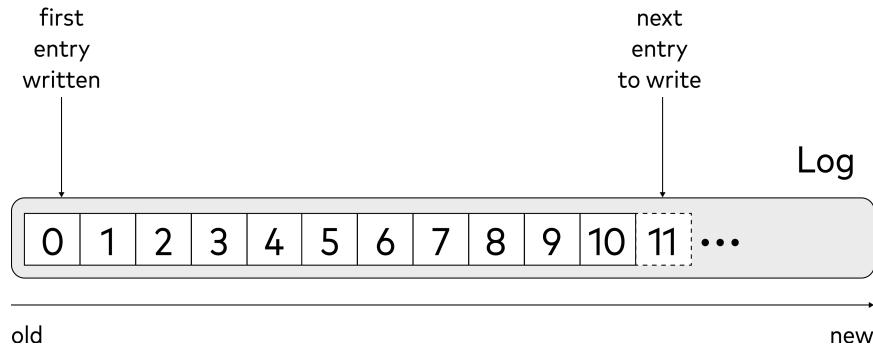
Let's talk about the entities that produce data to and consume data from the commit logs.

- **Producers:** these are applications that write data to one or many topics.
- **Consumers & Consumer Groups:** Consumers are organized in consumer groups. Members of a consumer group collaborate and as such allow the parallel processing of data. Consumers in a consumer group are instances of the same application with the same application ID.

Producers and consumers are both called **Kafka clients**.

At a high-level, Kafka is a pub-sub messaging system that has producers that capture events. Events are sent to and stored locally on a central cluster of brokers. And consumers subscribe to topics or named categories of data.

# The Kafka Commit Log



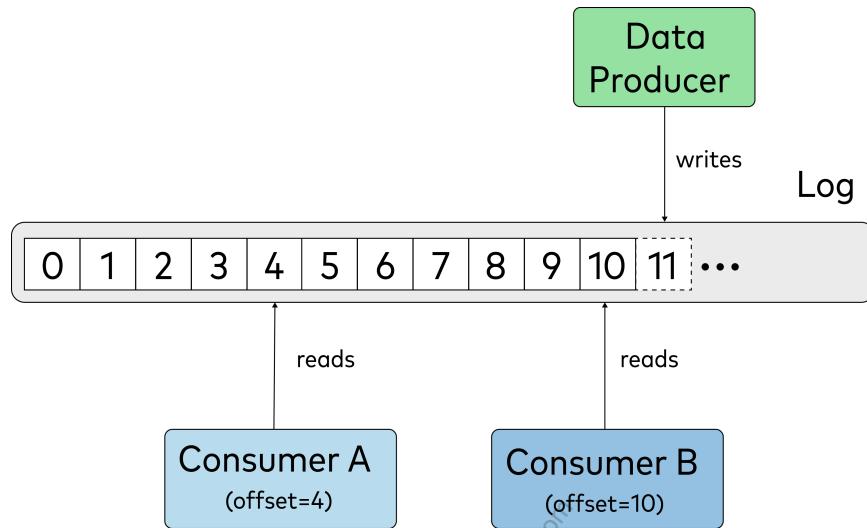
A position in the log is called an "offset." Here we see offsets 0 through 11.

To understand Kafka as a **Streaming Platform**, it is important that we first discuss a few basics. One central element that enables Kafka and stream processing is the "log" abstraction, also known as the "commit log".

A log is an immutable, append-only data structure, which means data elements are always appended to the end of the log and never changed.

We can often think of a log as a stream of events where events happen over time. The time axis on the image helps to reinforce this idea. The position of an event in the log is called an "offset". Offsets are closely related to timestamps. Time is a delicate subject in any distributed system, so different notions of time relevant to Kafka are discussed more deeply in upcoming modules.

# Decoupling Data Producers from Data Consumers



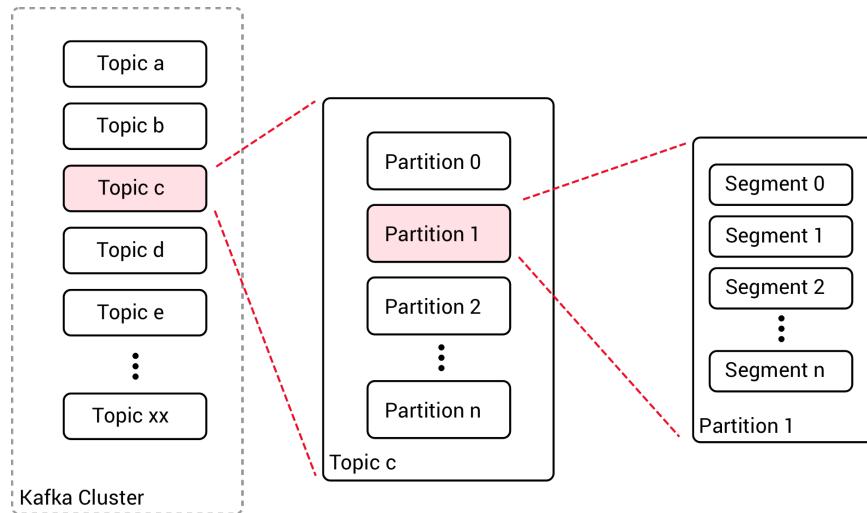
The log is produced by some data source. In the image the data source has already produced elements with index 0 to 10, and the next element the source produces will be written at index 11.

The data source can write at its own speed since it is totally decoupled from any destination system. In fact, the source system does not know anything about the consuming applications.

Multiple destination systems can independently consume from the log, each at its own speed. In the sample we have two consumers that read from different positions at the same time. This is totally fine and an expected behavior.

Each destination system consumes the elements from the log in temporal order, that is from left to right in our image.

# Kafka—Logical View of Topics, Partitions, and Segments

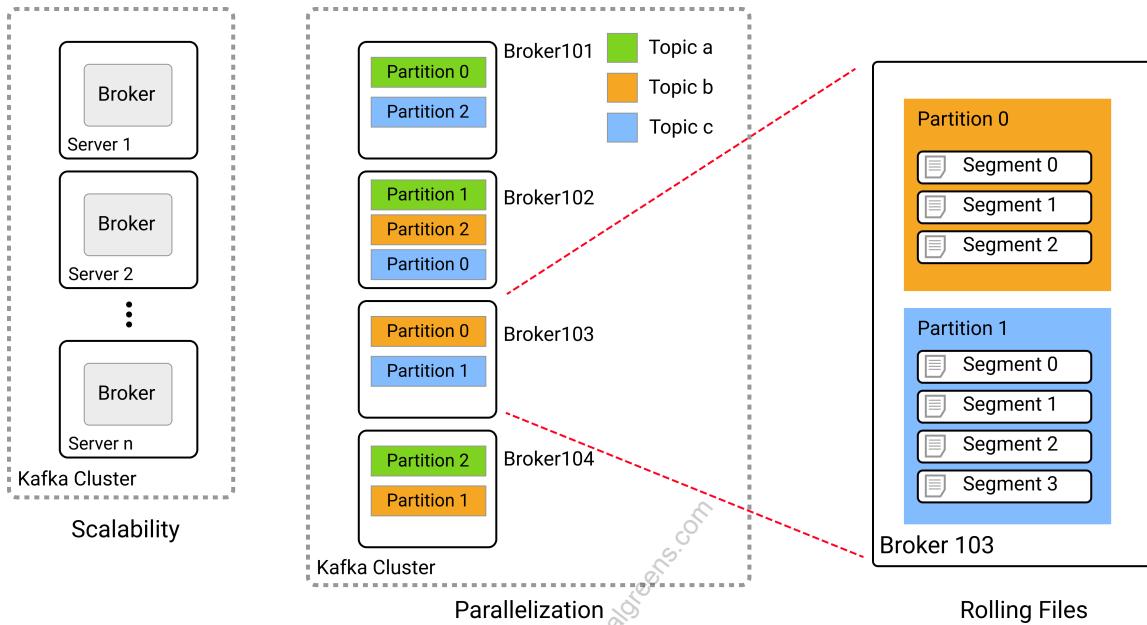


Each partition is a commit log. We often use "partition" and "log" interchangeably.

Kafka implements logs in terms of **Topics, Partitions and Segments**:

- **Topic:** A topic is a set of related messages. We could have a topic called "driver-positions" where each message is a driver's latitude and longitude at a certain time. We will discuss what exactly a "message" is in an upcoming slide.
- **Partition:** To parallelize work and thus increase the throughput, Kafka can split a single topic into many partitions. The messages of the topic will then be balanced across the partitions. A partition is Kafka's implementation of the "commit log." Determining the number of partitions for a topic, and how data is balanced across those partitions, is critical to designing performant applications. Partitioning will be discussed several times throughout the course.
- **Segment:** Data written to a partition is stored in a physical file called a log segment file. Since the data can potentially be endless, Kafka uses a "rolling-file" strategy. Kafka allocates a new file and fills it with messages. When the log segment has gotten larger than a specified size or older than a specified age, a new file is opened and messages are now appended to the new file. The data in the partition grows and grows over time, but Kafka will delete segment files older than a specified age to keep from filling up storage.

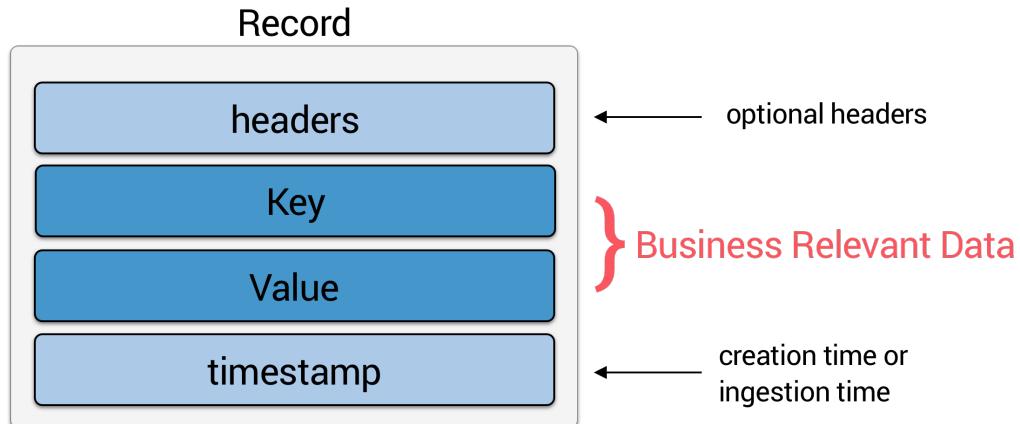
# Kafka—Physical View of Topics, Partitions and Segments



To put it simply, a Kafka broker is software that manages reads and writes for partitions. A Kafka cluster is a collection of one or more brokers. The more servers running as Kafka brokers there are, the more resources are available for storing and managing read/write requests to partitions. If we have three topics (that is 3 categories of messages), then they may be organized physically as shown in the graphic.

- Partitions of each topic are distributed among the brokers
- Each partition on a given broker results in one to many physical files called segments
- Segments use a rolling file strategy. Kafka allocates a file for the segment and appends to it until it closed. Kafka closes a segment file if it grows past a certain size (configured by `log.segment.bytes`, default 1GB) or the newest message becomes older than a certain age (configured by `log.roll.hours`, default 168 hours). Subsequently a new segment file is created for the log (also known as "rolling the log").

# The Record—The Atomic Unit of Kafka



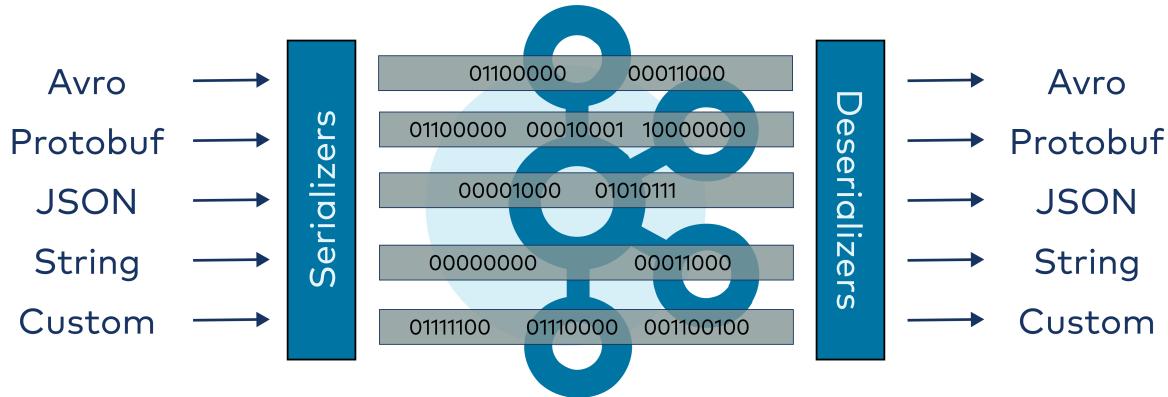
Kafka Records are also known as "Messages" or "Events"

A data element is called a **record** in the Kafka world. We often use **record**, **message**, and **event** synonymously.

- Headers are optional key-value arrays that allow a developer to tag messages with information that is not relevant to the business. For example, Confluent Replicator uses headers to tag the data center where a message first arrived into Kafka.
- The most important parts of the record are its key and value. This is the business relevant data. The key determines which Partition the record will be appended to, which is important for scalability. The value is the actual payload of information the business needs to process.
- Developers can configure the timestamp for a record to be either:
  - **CreateTime (default)**: when the client created the message, or
  - **LogAppendTime**: when the message is appended to the partition in Kafka.

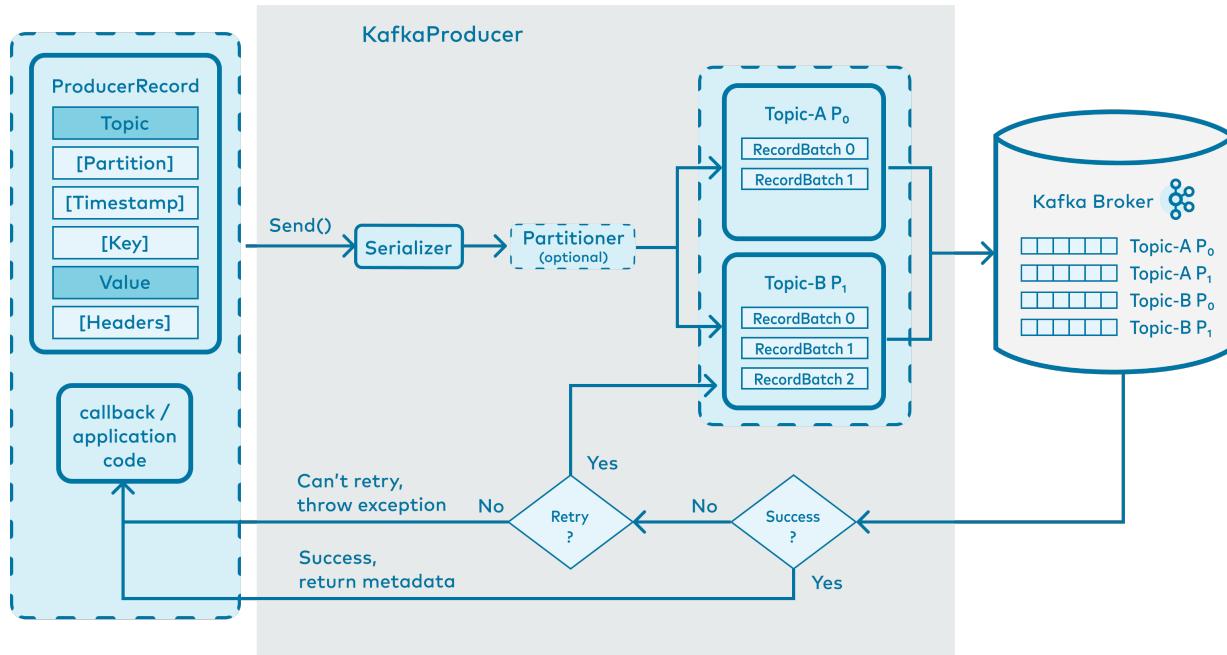
# Serialization

- Kafka stores byte arrays



Kafka stores data in arrays of bytes. That means producers must convert data from its native format into byte arrays before sending to Kafka. This conversion is called serialization. Conversely, the data consumers fetch from Kafka comes in the form of byte arrays that must be deserialized in order to be read.

# Producer Design



In this graphic you can see the high level architecture of a Kafka producer.

On the left hand side you see a **ProducerRecord** that is the data the producer wants to send to Kafka.

## ProducerRecord elements

- Topic - Required. A topic name to which the record is being sent
- Partition - Optional.
  - If a valid partition number is specified, that partition will be used when sending the record.
  - If no partition is specified but a key is present, a partition will be chosen using a hash of the key.
  - If neither key nor partition is present, a partition will be assigned in a round-robin fashion.
- Timestamp - Optional. If the client does not provide a timestamp, the producer will stamp the record with its current time. The timestamp eventually used by Kafka depends on the timestamp type configured for the topic.
  - If the topic is configured to use CreateTime, the timestamp in the producer record will be used by the broker.
  - If the topic is configured to use LogAppendTime, the timestamp in the producer

record will be overwritten by the broker with the broker local time when it appends the message to its log.

- Key - Optional.
- Value - The record contents.
- Headers - Optional. Key-value arrays.

Once your code sends the record it goes through the serializer and the partitioner. After those steps individual records are batched for improved throughput.

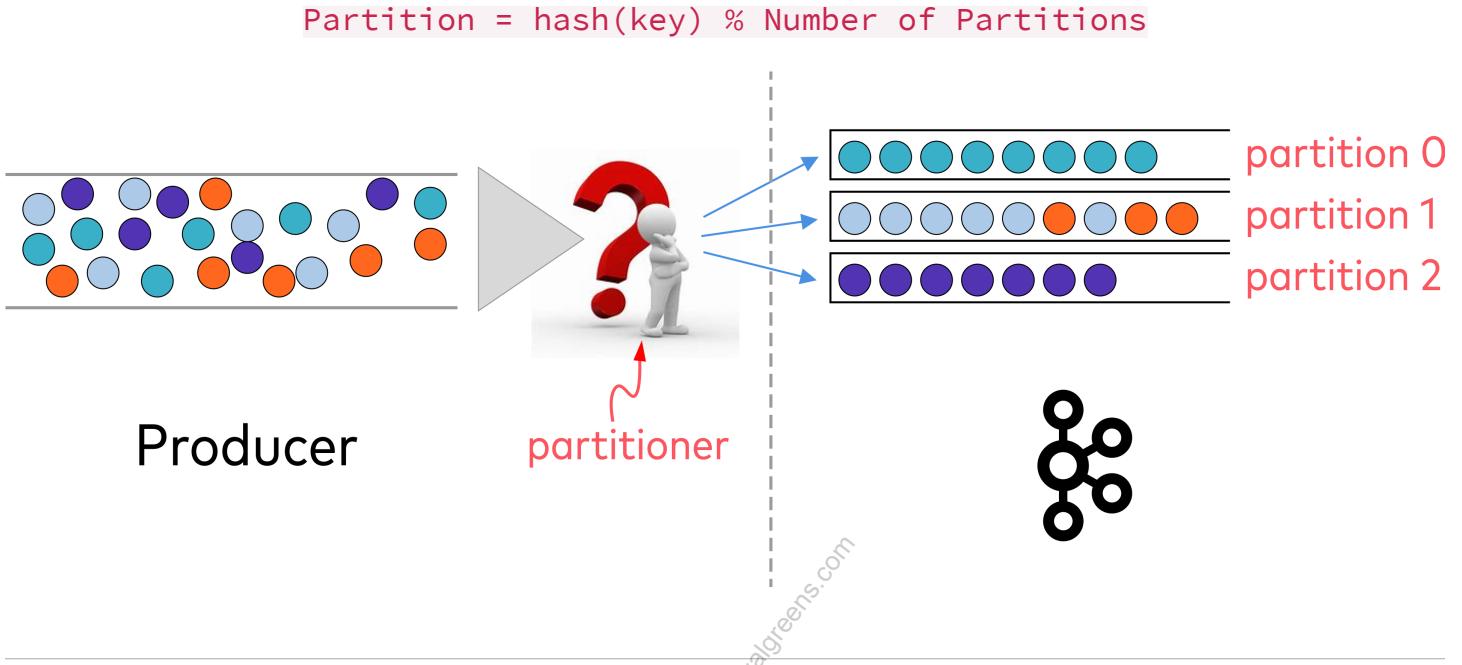
- Batching happens on a per topic partition level (i.e. only messages written to the same partition are batched together)

Optionally there might also be compression involved after the above steps - it is not shown here on this slide. The producer then tries to write the batch to the Kafka broker. The broker will answer either with ACK or NACK:

- If ACK, then all is good and success metadata is returned to the producer
- If NACK, then the producer transparently retries until a configurable timeout has been reached, in which case an exception is returned to the producer

For more details about the system metadata included in RecordBatch and Record objects, see <https://kafka.apache.org/documentation/#messageformat>.

# Partitioning



The Producer API includes a default partitioner that determines which Partitions each message should land on. By default, messages are partitioned by key according to the algorithm:

$\text{Partition} = \text{hash}(\text{key}) \% \# \text{Partitions}$

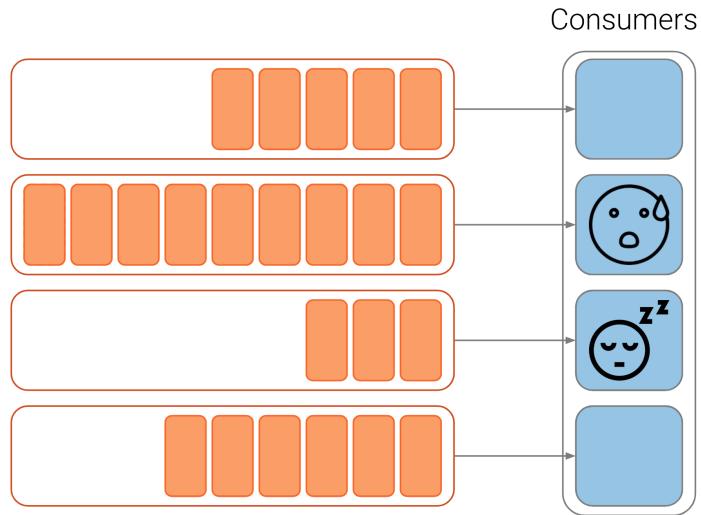
This means messages with a given key will always be sent to the same Partition, so long as the number of Partitions stays the same. This has some implications for software design. Partitioning is important to consider for:

- Aggregating or joining by key
- Guaranteeing message ordering on a per-key basis
- Scaling large quantities of data across many Brokers
- Scaling downstream Consumer processing



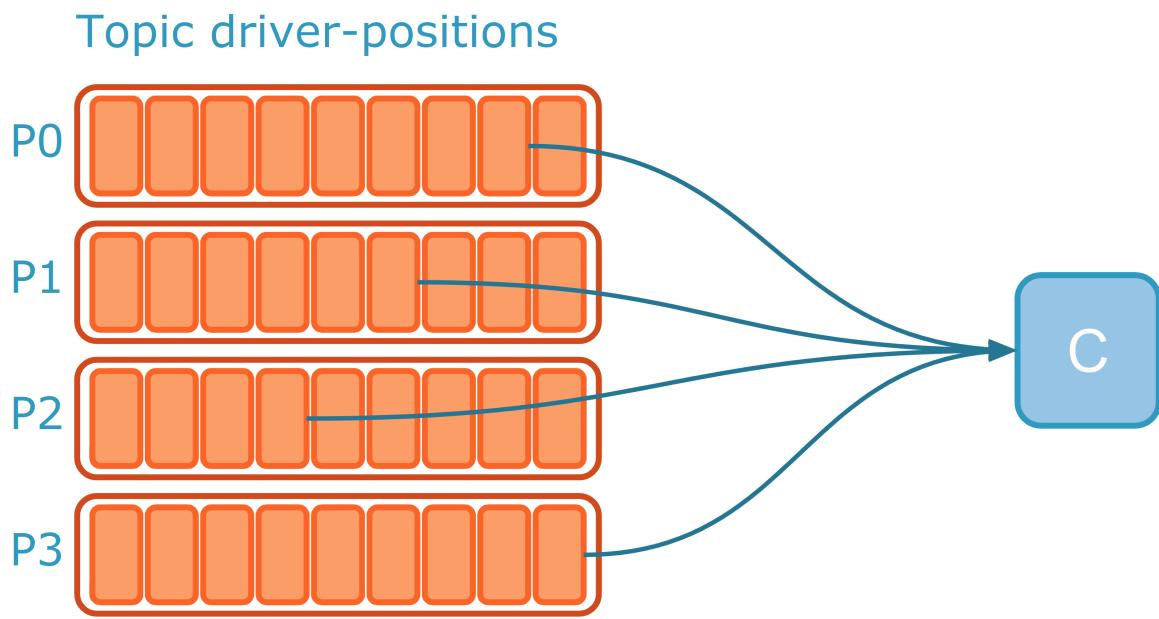
A custom partitioner can be added to the Producer code to override the default partitioner.

# Cardinality



- **Cardinality:** the number of elements in a set or other grouping, as a property of that grouping.
- Key cardinality affects the amount of work done by the individual Consumers in a group. Poor key choice can lead to uneven workloads.
- Keys in Kafka don't have to be simple types like Integer, String, etc. They can be complex objects with multiple fields. For example, in some cases, a compound key can be useful, where part of the key is used for partitioning, and the rest used for grouping, sorting, etc. So, create a key that will evenly distribute groups of records around the Partitions.

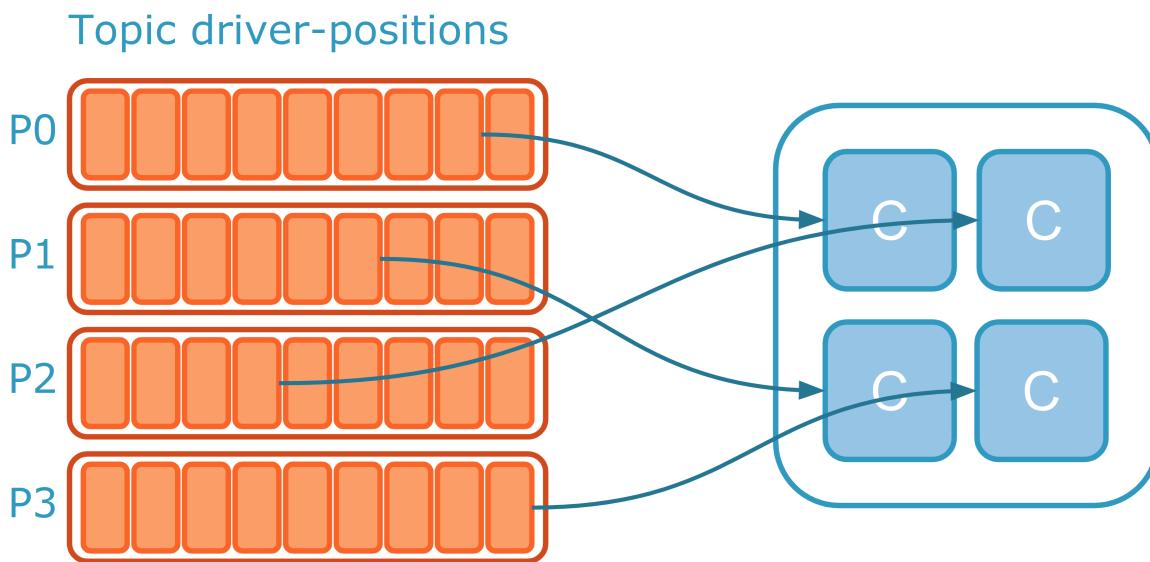
# Consuming from Kafka - Single Consumer



If you have a single consumer that consumes data from a topic, here with 4 partitions, then this consumer will consume all records from all partitions of the topic.

tejaswin.renugunta@datafans.com

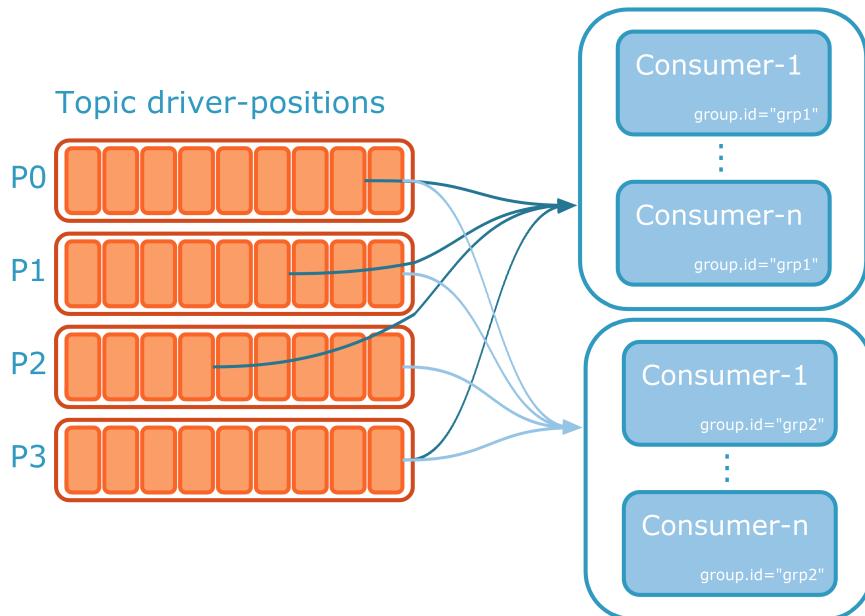
# Consuming as a Group



A consumer group transparently load balances the work among the participating consumer instances. In this image we have 4 consumers that consume a topic with 4 partitions. Thus each consumer consumes records from exactly one partition.

A partition is always consumed as a whole by a single consumer of a consumer group. A consumer in turn can consume from 0 to many partitions of a given topic.

# Multiple Consumer Groups



Any collection of consumers configured with the same `group.id` name will form a **consumer group**. The consumers in a consumer group will split up the workload among themselves in a somewhat even fashion. Note that, as indicated here on the slide, we can have multiple consumer groups consuming from the same topic(s).

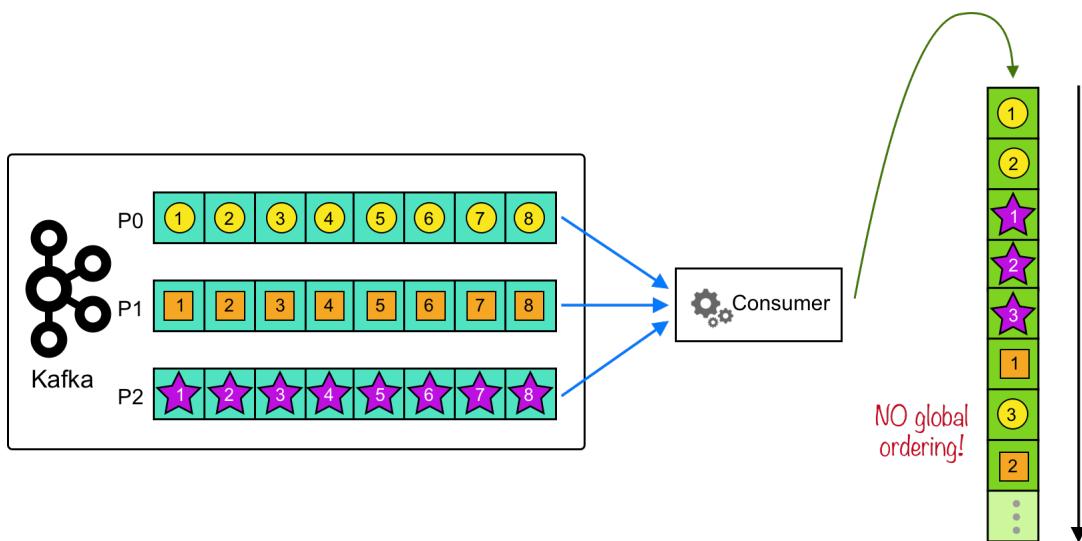
tejaswin.rengaraju@algens.com

# Preserve Message Ordering

- Messages with the same key, from the same Producer, are delivered to the Consumer in order
  - Kafka hashes the key and uses the result to map the message to a specific Partition
  - Data within a Partition is stored in the order in which it is written
  - Therefore, data read from a Partition is read in order *for that partition*
- If the key is null and the default Partitioner is used, the record is sent to a random Partition

tejaswin.renugunta@walgreens.com

# An Important Note About Ordering



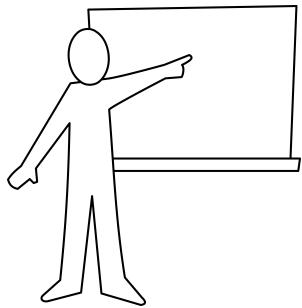
- **Question:** How can you preserve message order if the application requires it?

- If there are multiple Partitions, you **will not get total ordering across all messages** when reading data

Answer:

- Be selective when choosing the message key
  - Messages with the same key are delivered to the Consumer in order
- Use a single Producer
  - There are no guarantees that different Producers writing to the same Topic with the same key will send in order, due to batching, CPU, etc
- Ensure that the application calling the Kafka Producer is preserving message order as well
  - Send synchronously to the Producer, e.g. wait till the first message is received before sending the second
- Make the applications responsible for ordering outside of Kafka
  - Have Producers include information that can be used by the Consumers to reorder the messages after receipt

# Module Map



- Event Streaming Platform
- Event Streaming Architecture
- Replication ... ←
- Hands-on Lab: **Introduction**
- Hands-on Lab: **Using Kafka's Command-Line Tools**
- Hands-on Lab: **Consuming from Multiple Partitions**

tejaswin.renugunta@walgreens.com

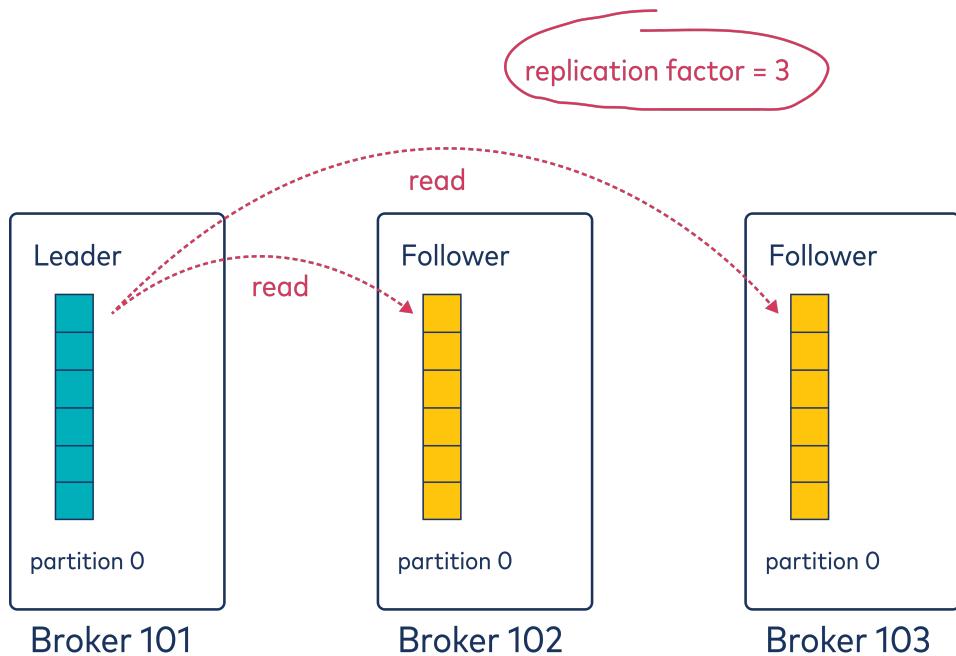
# Problems With our Current Model

- So far, we have said that each Broker manages one or more Partitions for a Topic
  - This does not provide reliability
    - A Broker failing would result in all of those Partitions being unavailable
  - Kafka takes care of this by replicating each Partition
    - The replication factor is configurable
- 

Partitioning data is good for performance but not for reliability. The more parts there are in any system, the higher the probability that one of them will fail. Kafka addresses this problem by included a built-in replication solution to maintain multiple copies of each partition.

tejaswin.renugunta@walgreens.com

# Replication of Partitions

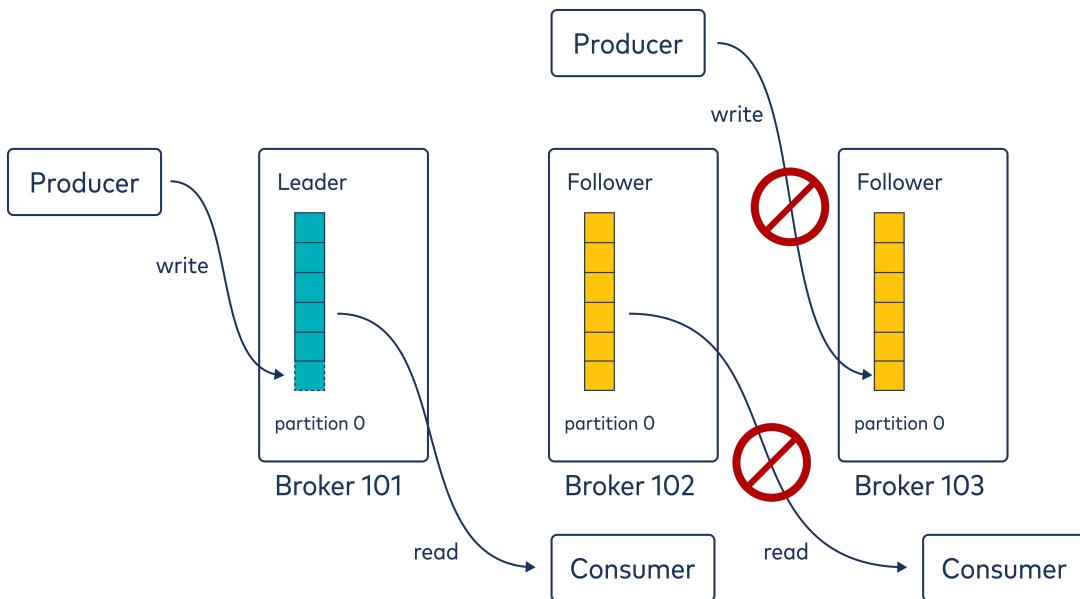


- Kafka maintains replicas of each partition on other Brokers in the cluster
  - Number of replicas is configurable
- One Broker is in the role of **leader** for a particular partition
  - All writes and reads go to and from the leader
  - Other Brokers are in the role of **follower** for that partition

At any given point of time, all the replicas are byte wise identical (except for where one hasn't caught up). To facilitate that, the ordering of the messages must be identical on all replicas.

For any given replicated partition, one replica is a leader and the rest are followers. All I/O (produce and consumer requests) go to the leader. In the case of write requests, this allows just one replica (the leader) to determine the ordering of messages. Once the messages are written to the local log of the leader, all the replicas can get the data.

# Clients Interact with Leaders



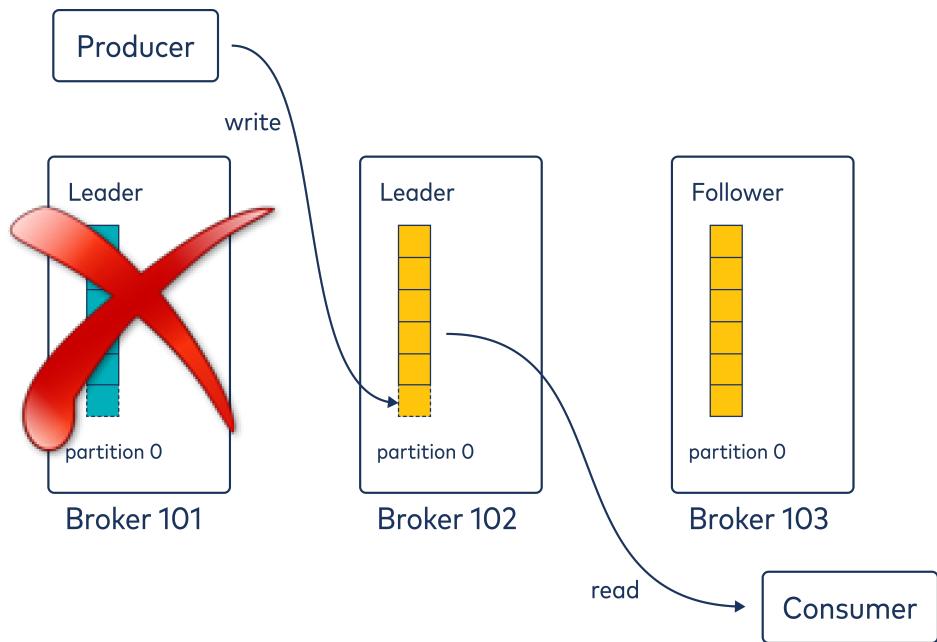
Caveat: Apache Kafka 2.4 introduced follower fetching for consumers.

- It is important to understand that Producers only write to the leader
  - This allows for ordering and consistency guarantees
- By default, Consumers also read from the leader
  - Consumers and brokers can be configured to allow consumers to read from followers ("follower fetching")
- Replicas only exist to provide reliability in case of Broker failure
- Followers copy the data from the commit log of the leader as a fetch request. They don't generally interact with clients (except consumers with follower fetching enabled)



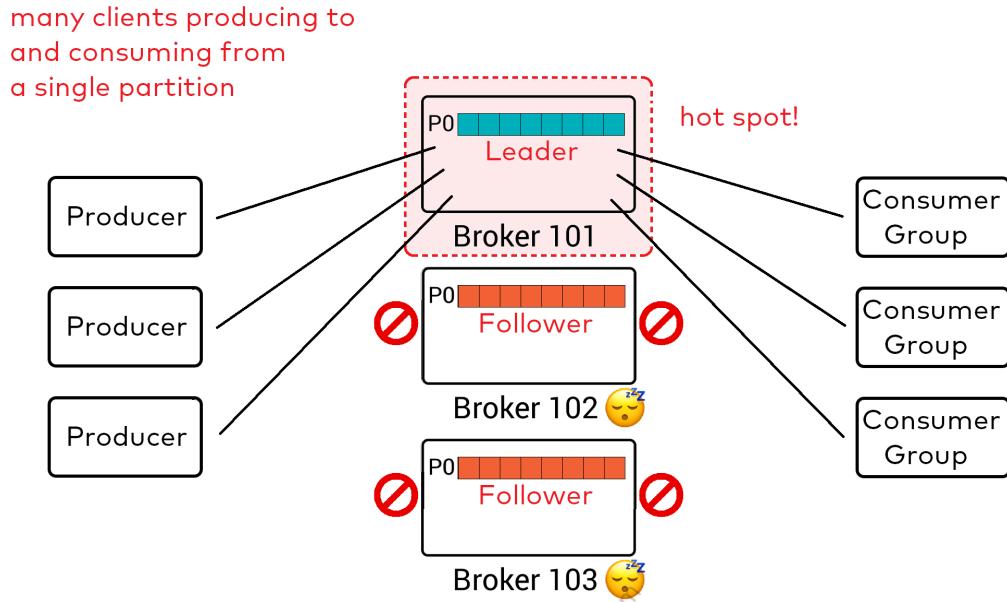
Follower fetching will be discussed in more detail in a later module.

# Leader Failover



- If a leader fails, the Kafka cluster will elect a new leader from among the followers
- The clients will automatically switch over to the new leader

# Load Balancing Partitions Leadership (1)

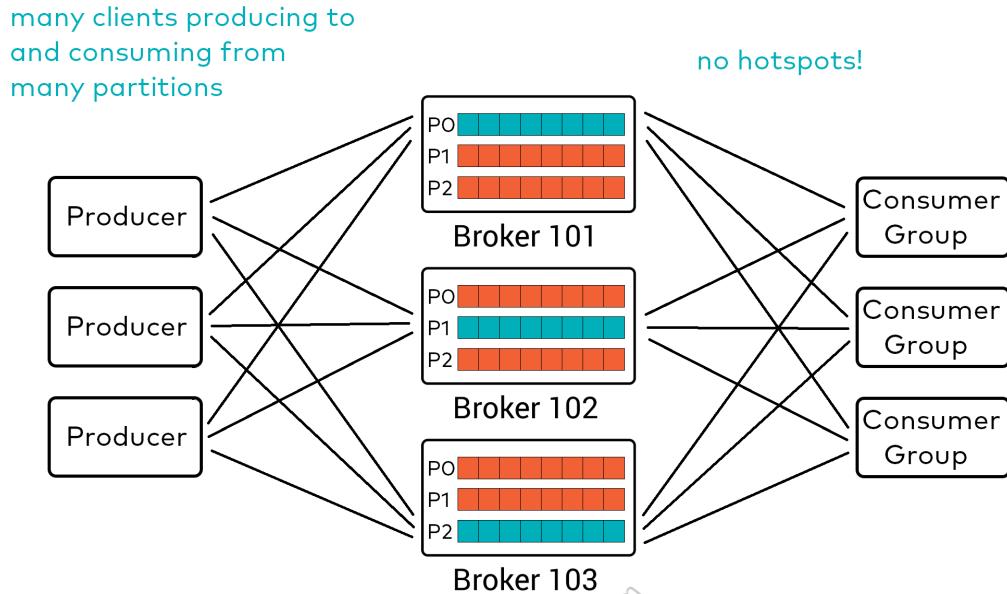


- **Recall:** By default, all consumers read from the leader of a partition
- Producers write only to the leader
- This would lead to "hot brokers" if partition leadership were not equally distributed among the brokers
- Luckily, Kafka has mechanisms for load balancing leadership across brokers, which we will see in the next slide



The followers are not sleeping. Rather, the purpose of the image is to emphasize that leadership imbalance leads to an imbalance of resource usage because leaders handle most client requests.

# Load Balancing Partitions Leadership (2)



In a multi-partition topic with replication, each partition will have its own leader. The leader of a partition serves more traffic than the followers since it handles all produce and most fetch (consume) requests. Kafka will spread the leaders across the available brokers to balance the load.

A broker is usually both a leader and a follower, depending on which partition you are talking about.

Both producers and consumers need to know which Broker is the leader for the partition they need to contact. This is done through a metadata request. When the client starts, it does not know which brokers are the leaders of which partitions, so it issues a metadata request to a broker (any Broker). Partition information is cached on all the Brokers, so any Broker can respond to a metadata request. After receiving the updated metadata, the client will know who the leader is for each partition and will communicate with the appropriate brokers. If a broker fails, the client will get an error and will refresh its metadata.

# Hands-On Lab

- In this Hands-On Exercise, you will run a simple Kafka cluster and use some of the Kafka command line tools to do basic operations on the cluster.
- Please refer to **Lab 02 Fundamentals of Apache Kafka** in the Exercise Book:
  - a. **Introduction**
  - b. **Using Kafka's Command-Line Tools**
  - c. **Consuming from Multiple Partitions**



tejaswin.renugunta@walgreens.com

# Module Review



## Questions:

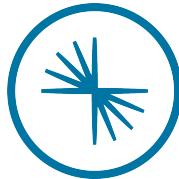
1. Explain the relationships between **records**, **topics**, and **partitions**.
2. How does Kafka achieve **high availability**?
3. What is the **easiest** way to get data from a database into Kafka?

---

## Answers:

1. Topics are logical groupings of messages in Kafka and (usually) contain records of the same type. Partitions are used to parallelize work with topics. Thus one topic has 1...n partitions.
2. Kafka uses replication to achieve high availability. Each partition of a topic is available in the cluster on multiple distinct brokers based on the replication factor. If one broker fails, then new leaders will be elected for those lead partitions that were lost.
3. Kafka Connect is the easiest way to get data into or extract data from Kafka. For example, the JDBC source connector is a great way to get database tables into Kafka topics. Conversely, the JDBC sink connector is a great way to pump Kafka records into database tables.

# 03: Providing Durability



# CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Kafka Fundamentals
3. Providing Durability ... ←
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance
6. Kafka Security
7. Data Pipelines with Kafka Connect
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

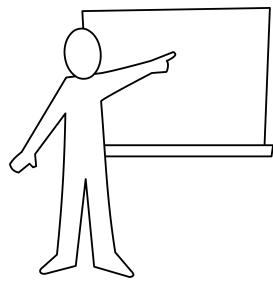


After this module you will be able to:

- Describe data replication in the Kafka Cluster
- Explain how the cluster recovers from failures
- Explain how Producers reliably send messages to the Brokers
- Explain how offset management impacts the end user application
- Explain how Kafka achieves Exactly Once Semantics (EOS)

tejaswin.renugunta@walgreens.com

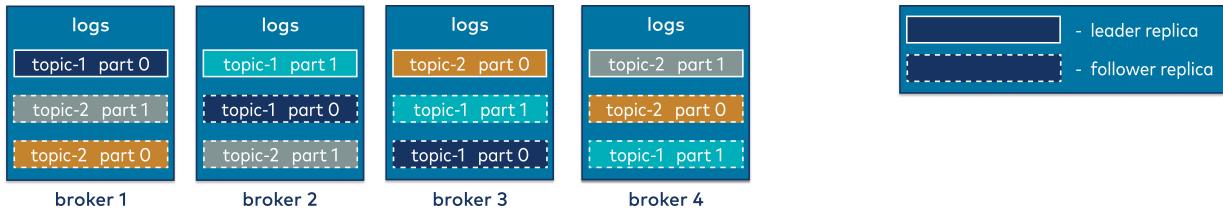
# Module Map



- Replication Review ... ↵
- Advanced Replication Concepts
- Writing Data Reliably
- Broker Shutdowns and Failures
- The Kafka Log Files
- Offset Management
- Exactly Once Semantics (EOS)
- 🔧 Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

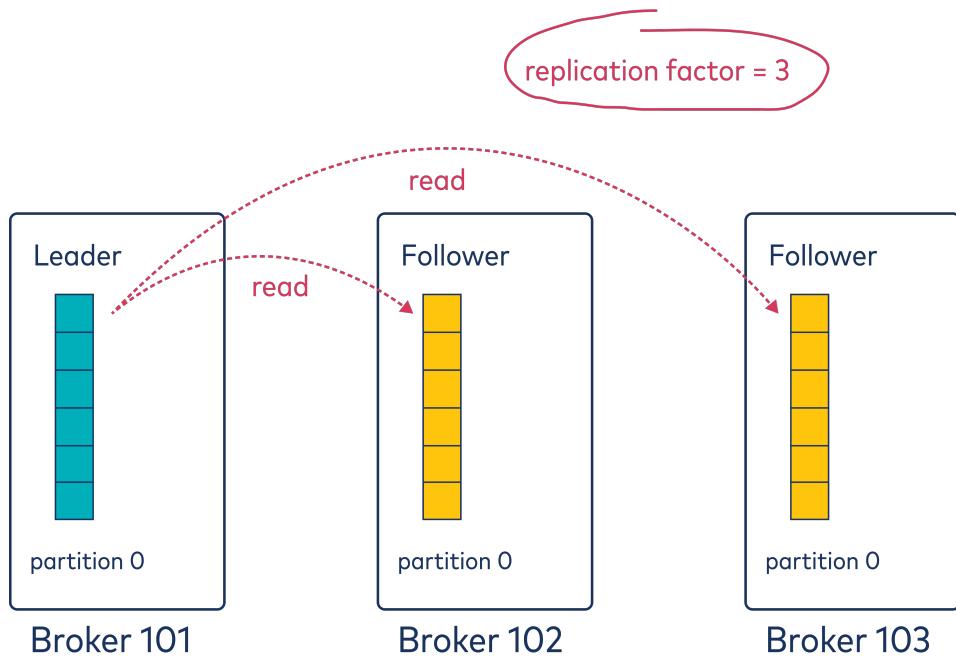
# Review - Partition Replicas



- How many Topics does this cluster have?
- What is a Partition and how many Partitions does each Topic have?
- What is the replication factor of **topic-1**?
- What do "leader" and "followers" mean in the context of Partition replication?
- What would this cluster do if Broker 2 failed?
- What would this cluster do if a new Topic was created with 1 Partition and a replication factor of 12?

tejaswin.renugunta@walgreens.com

# Review—How Replication Works

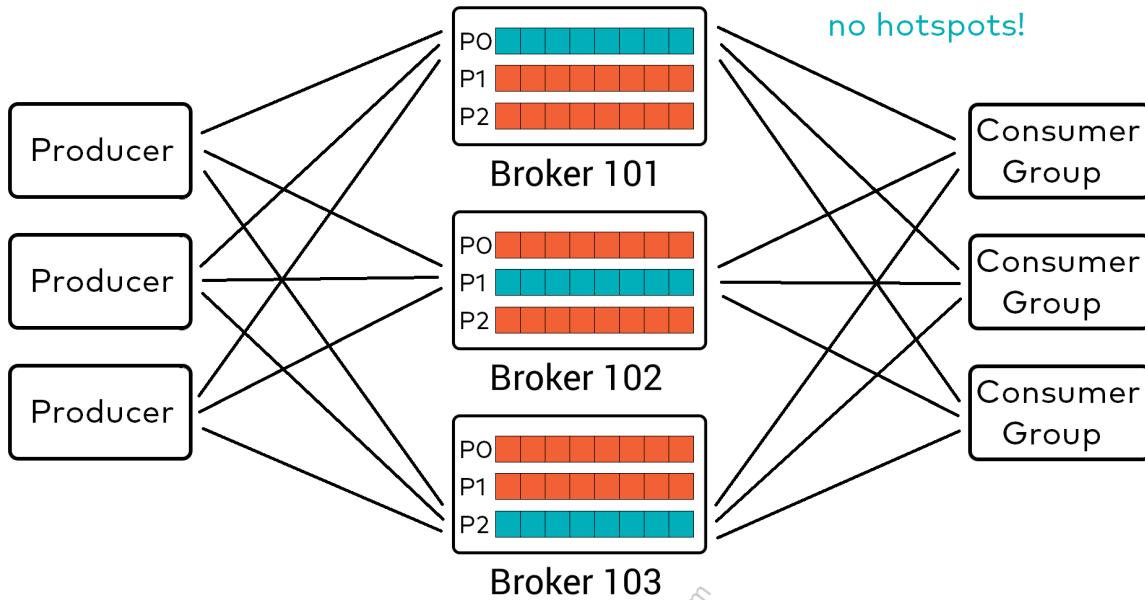


At any given point of time, all the replicas are byte-wise identical (except for where one hasn't caught up). To facilitate that, the ordering of the messages must be identical on all replicas.

For any given replicated Partition, one replica is designated the leader for the Partition and the other replicas are designated as followers. All I/O (Producer and Consumer requests) go to the Broker with the leader replica. This allows just one replica (the leader) to determine the ordering of messages. Once the messages are written to the local log of the leader, all the replicas can get the data. The followers just copy the data from the commit log of the leader as a pull request; they do not interact with any external clients.

# Review—Load Balancing Partition Leaders

Leaders are load balanced to avoid "hot Brokers"



In a multi-Partition Topic with replication, each Partition will have its own leader. The leader of a Partition serves more traffic than the followers since it handles all produce and consume (or fetch) requests. Kafka will spread the leaders across the available Brokers to balance the load.

# Review—Replica Leaders and Followers

For a given Partition:

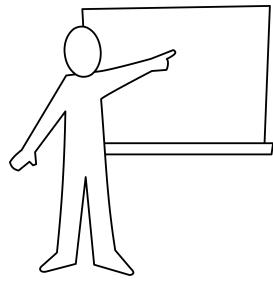
- Leader
    - Accepts all writes and reads for a specific Partition
    - Monitors health of follower replicas
  - Followers
    - Attempt to keep up with the leader
    - Provide fault tolerance
  - Leader election rate (JMX metric)  
`kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs`
- 

The leader is the only replica that communicates with clients. We will discuss leader failover later in this module.

Followers can be thought of as specialized Consumers. The whole job of a follower is to copy the contents of the commit log on the leader onto itself.

One way to determine the health of a cluster is to monitor how often leaders are being elected. A non-zero value for the `LeaderElectionRateAndTimeMs` metric indicates that leaders are being reelected, which typically is a symptom of Broker or network failures within the cluster.

# Module Map



- Replication Review
- Advanced Replication Concepts ... ↵
- Writing Data Reliably
- Broker Shutdowns and Failures
- The Kafka Log Files
- Offset Management
- Exactly Once Semantics (EOS)
- 🔧 Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

# Replica Configurations

- Increase the replication factor for better durability guarantees
- For auto-created Topics
  - `default.replication.factor` (Default: 1)
  - Configure in `server.properties` on each Broker
- For manually-created Topics:

```
$ kafka-topics --bootstrap-server kafka-1:9092 \
  --create --topic my_topic \
  --replication-factor 3 \
  --partitions 2
```

---

By default, Kafka Brokers allow automatic Topic creation. If a Topic is automatically created, it will be configured according to the defaults in the `server.properties` file of the Brokers. The default setting for `default.replication.factor` is 1 - change it to an appropriate value for your environment. For production environments, it is suggested to use a replication factor of at least 3 (and thus, you would need at least 3 Brokers).

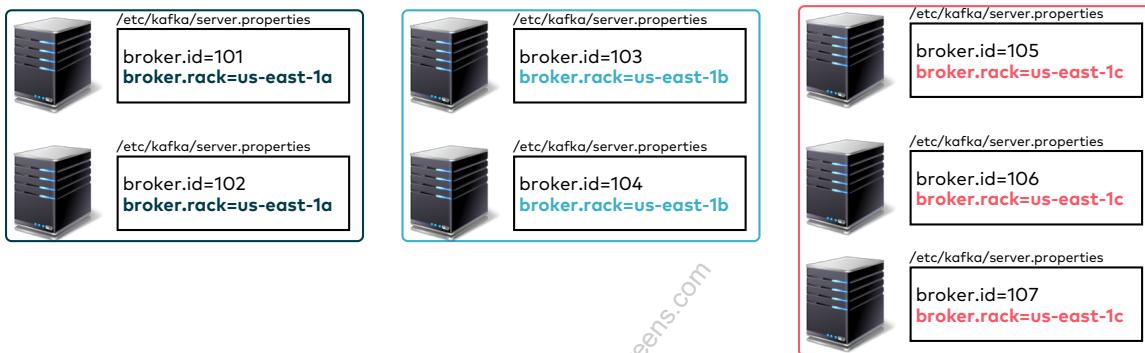
If manually creating a Topic, you can specify the number of Partitions and replication factor. Notice that we do not specify which Brokers to use - in most cases it is best to let Kafka decide where to place replicas.

There are two other ways to create Topics:

- Clients: the Producer and Consumer applications can create Topics if the admin API library is added to the code. This library is available in the standard Java client and the TopicAdminAPI for C, Python, and Golang clients.
- Confluent Control Center: As of Confluent Platform 5.0, Control Center has the ability to create and manage Topics.

# Replicas Can Be Rack Aware

- All Broker properties are configured in `/etc/kafka/server.properties`
- One such property is **Rack Awareness**:
  - Specify the same "rack name" for Brokers in the same Availability Zone
  - Replicas will be balanced across racks with best effort
  - Only enforced on Topic creation or with Confluent Auto Data Balancer
  - Feature is **all or nothing**



In the previous slide, we mentioned setting the default replication factor on Brokers in the `server.properties` file. It is good to mention at this point that actually all configurable Broker properties are set in `/etc/kafka/server.properties`.

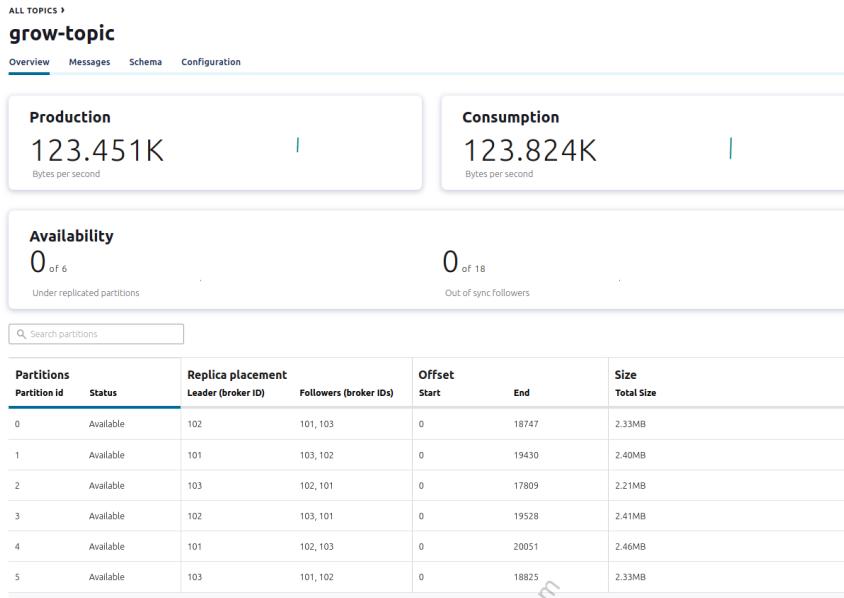
Rack awareness enforces replica placement across sets of Brokers to ensure resiliency in the face of an availability zone outage or rack failure. This is useful if deploying Kafka on Amazon EC2 instances across availability zones in the same region, or if grouping Brokers that share physical rack space in an on-premise data center. Specify the same "rack name" for Brokers in the same availability zone.

Rack awareness is only enforced on Topic creation and Auto Data Balancer operations (Auto Data Balancer will be discussed in more detail in an upcoming module). Setting rack awareness has no automatic effect on existing Topics.

If some, but not all Brokers have `broker.rack` set, then automatic Topic creation will ignore rack information and manual Topic creation will fail. Use `--disable-rack-aware` to force creation (with no rack awareness).

# Partition Placement Across Cluster (1)

Confluent Control Center provides per-Topic replica view:



Here is a Topic called **grow-topic** with 12 Partitions (0 through 11) and replication factor 3. Confluent Control Center helpfully shows where replicas are placed and whether they are out of sync.

A Broker is usually both a leader and a follower, depending on which Partition you are talking about. For example, Broker 102 in the image is the leader for Partitions 1 and 4, but a follower for Partitions 0, 2, and 3.

Note: Only 5 of 12 partitions of the grow-topic are visible in the partition list view on the slide. The list allows one to scroll the remaining partitions and their details into view.

# Partition Placement Across Cluster (2)

- The same data tracked from the CLI with:

```
$ kafka-topics --bootstrap-server kafka-1:9092 \
  --describe \
  --topic i-love-kafka

Topic:i-love-kafka PartitionCount:3      ReplicationFactor:3 Configs:
  Topic: i-love-kafka Partition: 0        Leader: 101 Replicas: 101,102,103  Isr:
  101,102,103
  Topic: i-love-kafka Partition: 1        Leader: 103 Replicas: 103,101,102  Isr:
  103,101,102
  Topic: i-love-kafka Partition: 2        Leader: 102 Replicas: 102,103,101  Isr:
  102,103,101
```



Preferred replicas highlighted in bold

The first element of the ISR is called the "**preferred replica**" for the leader of a Partition. Having a preferred replica allows the cluster to keep leader Partitions spread out amongst the Brokers.

tejaswin.renugunta@walmartlabs.com

# The Controller

- One Broker in the entire cluster is designated as the **Controller**
  - Controller monitors Broker liveliness via ZooKeeper
  - Controller communicates leader and replica information to other Brokers
  - Controller facilitates leader election
  - Pushes Partition metadata to ZooKeeper and Brokers
- Monitoring the Controller
  - `kafka.controller:type=KafkaController, name=ActiveControllerCount`
  - Alert if more than one Controller detected

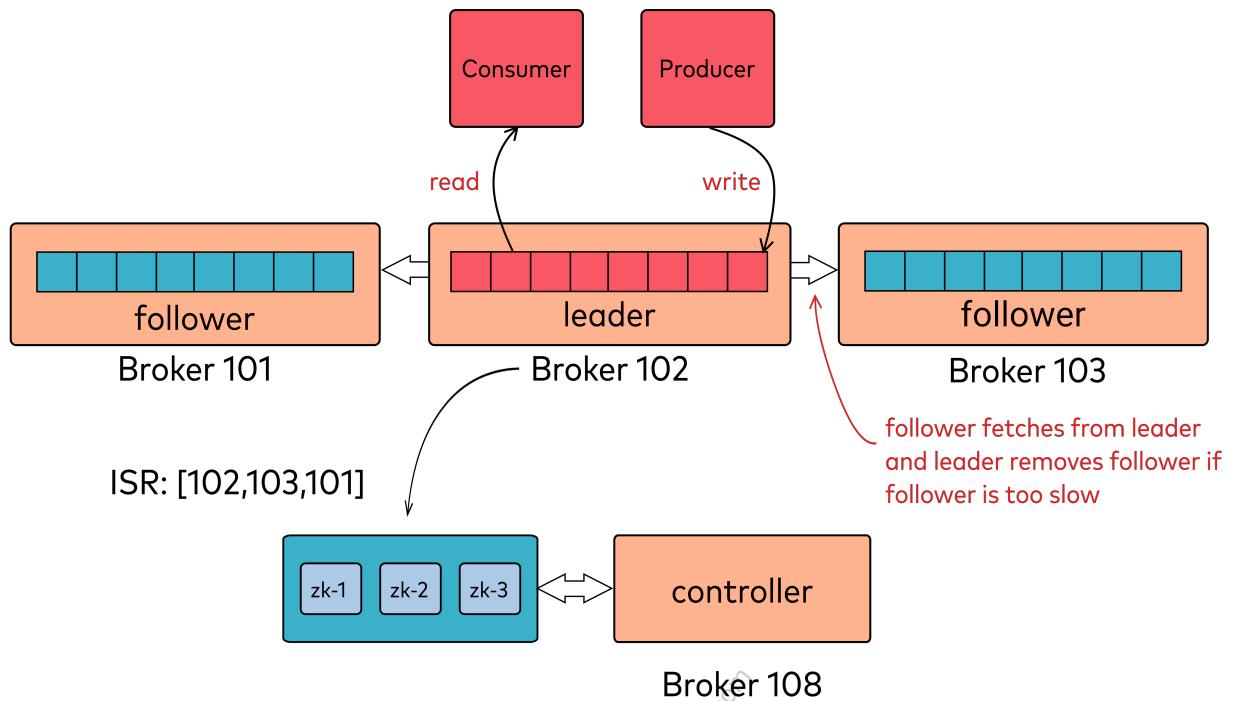
---

Managing the current list of leaders and followers (ISR list) for every replicated Partition is a full-time and mission-critical job. This task is handled by a special thread of the Broker software called the Controller. The Controller runs on exactly one Broker in the cluster, regardless of cluster size. It keeps a copy of the replica information in ZooKeeper and then caches the same information on every Broker in the cluster for faster access.

The Controller monitors the health of every Broker by monitoring their interaction with ZooKeeper. The Broker setting `zookeeper.session.timeout.ms` determines how long a Broker can be offline before it is considered dead (default 18 seconds). The Broker's zookeeper client sends a ping to zookeeper at an interval that is about 1/3 of `zookeeper.session.timeout.ms`. If a Broker registration in ZK goes away, that indicates a Broker failure. The Controller elects new leaders for Partitions whose leaders are lost due to failures and then broadcasts the metadata changes to all the Brokers so that clients can contact any Broker for metadata updates. The Controller also stores this data in ZooKeeper so that if the Controller itself fails, the new Controller will have a set of data to start with.

If the Controller fails, its associated `/controller` ephemeral node in ZooKeeper will disappear. Other Brokers in the cluster will be notified through the ZooKeeper watch that the Controller is gone and will attempt to create the Controller ephemeral node in ZooKeeper themselves. The first Broker to create the node successfully becomes the new Controller, while the other Brokers will receive a "node already exists" exception and re-create the watch on the new `/controller` node.

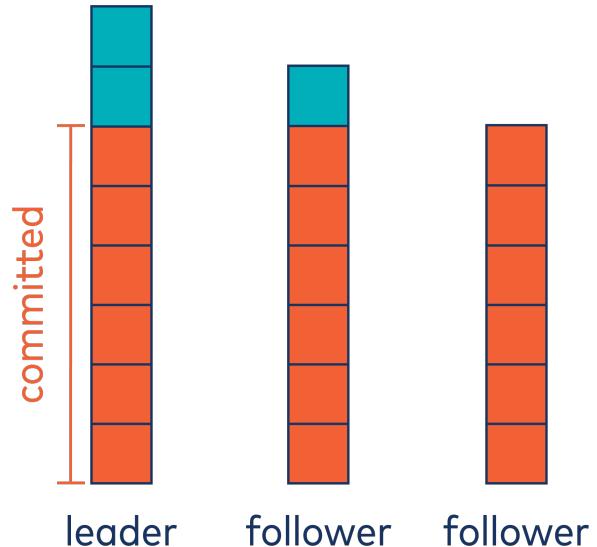
# In-Sync Replicas



- We have seen that Kafka uses replication to guarantee data durability. Each partition can be replicated. Each replica is on a different broker. There is one leader and the other replicas are followers.
- The In-Sync Replicas (ISR) is a list of the replicas - both leader and followers - that are identical up to a specified point called the **high-water mark**.
- If followers are too slow as determined by the `replica.lag.time.max.ms` setting, the leader removes them from the ISR and persists the ISR in ZooKeeper.
- The Controller listens to ZooKeeper for changes to Partition metadata (more on this later).
- If the leader fails, it is the list of ISRs which is used by the Controller to elect a new leader.

# What Does "Committed" Mean?

- A message is called "committed" when it is received by all the replicas in the ISR list
- Consumers can only read committed messages
- The leader decides when to commit a message
  - Committed state is checkpointed to disk



A message is marked committed by the leader if all of the in-sync replicas have fetched the message successfully. A committed message is guaranteed to have the same offset number on all of the followers. This means that no matter which replica is the leader (in the event of a failure), any Consumer will see the same data in that offset number. This is why a response to a Consumer's fetch request can only contain committed messages—this is how Kafka can make its data guarantees.

To make sure that the Broker retains a list of committed messages over restarts, the last committed offset for every Partition on the Broker is checkpointed to disk in a file called **replication-offset-checkpoint**. This file will be described in more detail later in this chapter.

# Synchronizing Replicas

- "High Water Mark"
  - Tracks offset of most recently committed message
  - Checkpointed to disk in `replication-offset-checkpoint`
- "Leader Epoch"
  - Marks offsets where new leaders are elected
  - Used during Broker recovery to truncate messages to a checkpoint and then follow the current leader
  - Checkpointed to disk in `leader-epoch-checkpoint`

---

Brokers have two mechanisms to ensure replicas are synchronized, avoid data loss, and prevent log divergence:

- The High Water Mark: the last committed offset for this replica
- Leader Epoch: shows the offset the last time the leader of the replica changed (more on this in upcoming slides about Broker failure)

During normal operations, the leader maintains the high water mark - the offset of the last committed message - and propagates this information to all replicas. Every replica persists the value to a checkpoint file on disk. The interval of writing to `replication-offset-checkpoint` is determined by the configuration value of `replica.high.watermark.checkpoint.interval.ms` (Default: 5 seconds).

Brokers using versions 0.11.0 and later rely on the leader to choose which of the markers is used to start replication in a recovery situation. Leader epoch is described in KIP 101:

<https://cwiki.apache.org/confluence/display/KAFKA/KIP-101+-+Alter+Replication+Protocol+to+use+Leader+Epoch+rather+than+High+Watermark+for+Truncation>

When the Controller elects a new leader, it updates the leader epoch and sends that information to all the members of the ISR list.

# Committing Messages with Replicas (1)



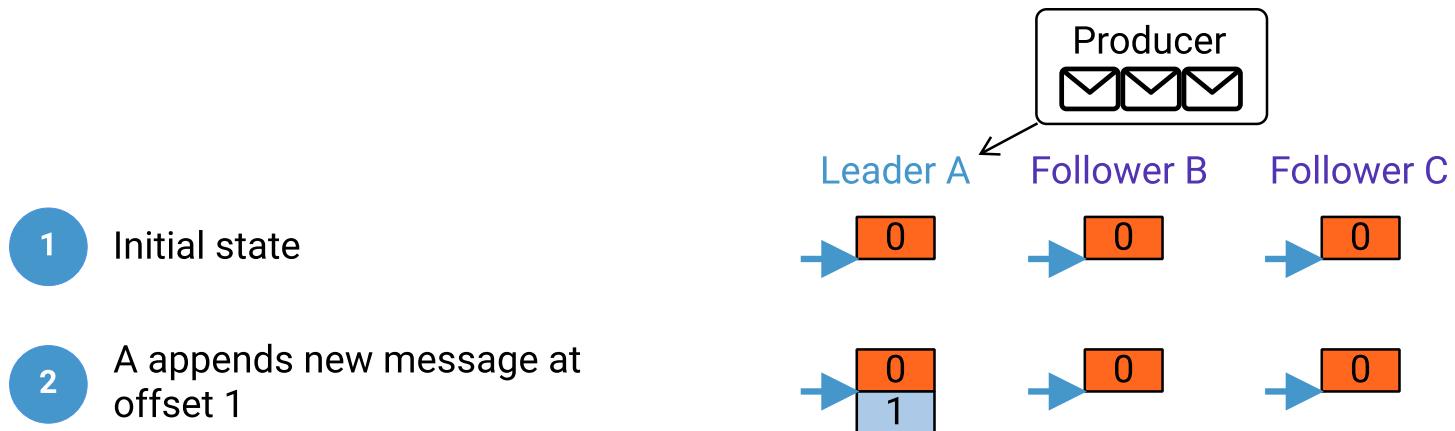
The next few slides walk through a typical replication/commit process.

Initially, the replicas for this Partition have been assigned with the leader on Broker A and followers on the other two Brokers. A message has been written to offset 0 on the leader and replicated to all followers. Therefore, the message at offset 0 is marked committed and the high water mark is set to offset 1.

But how does the leader know when the other replicas have received the message?

Traditional networking would use **ack** messages. However, that would add significant load to our networks if every replica is sending an **ack** for every message it receives. Kafka will use a more elegant approach that makes use of how followers fetch data.

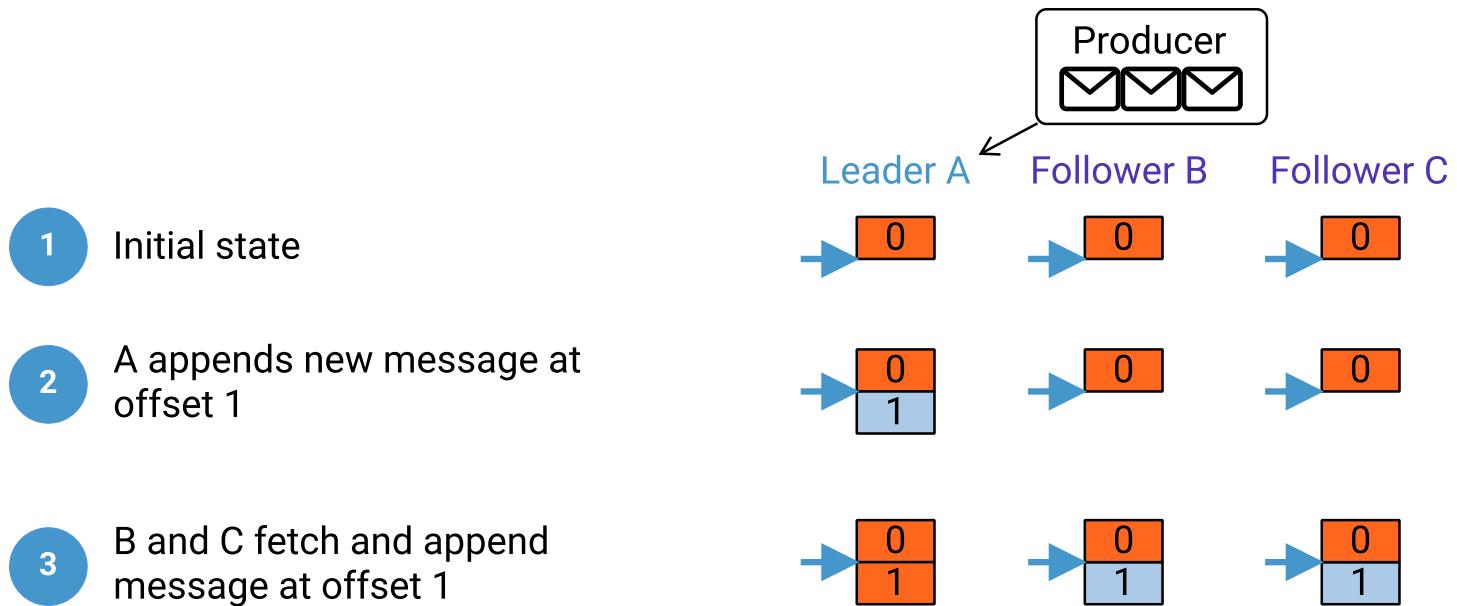
# Committing Messages with Replicas (2)



A message is received by the leader and written into offset 1. At this point, the followers have not requested the new data.

tejaswin.renugunta@walgreens.com

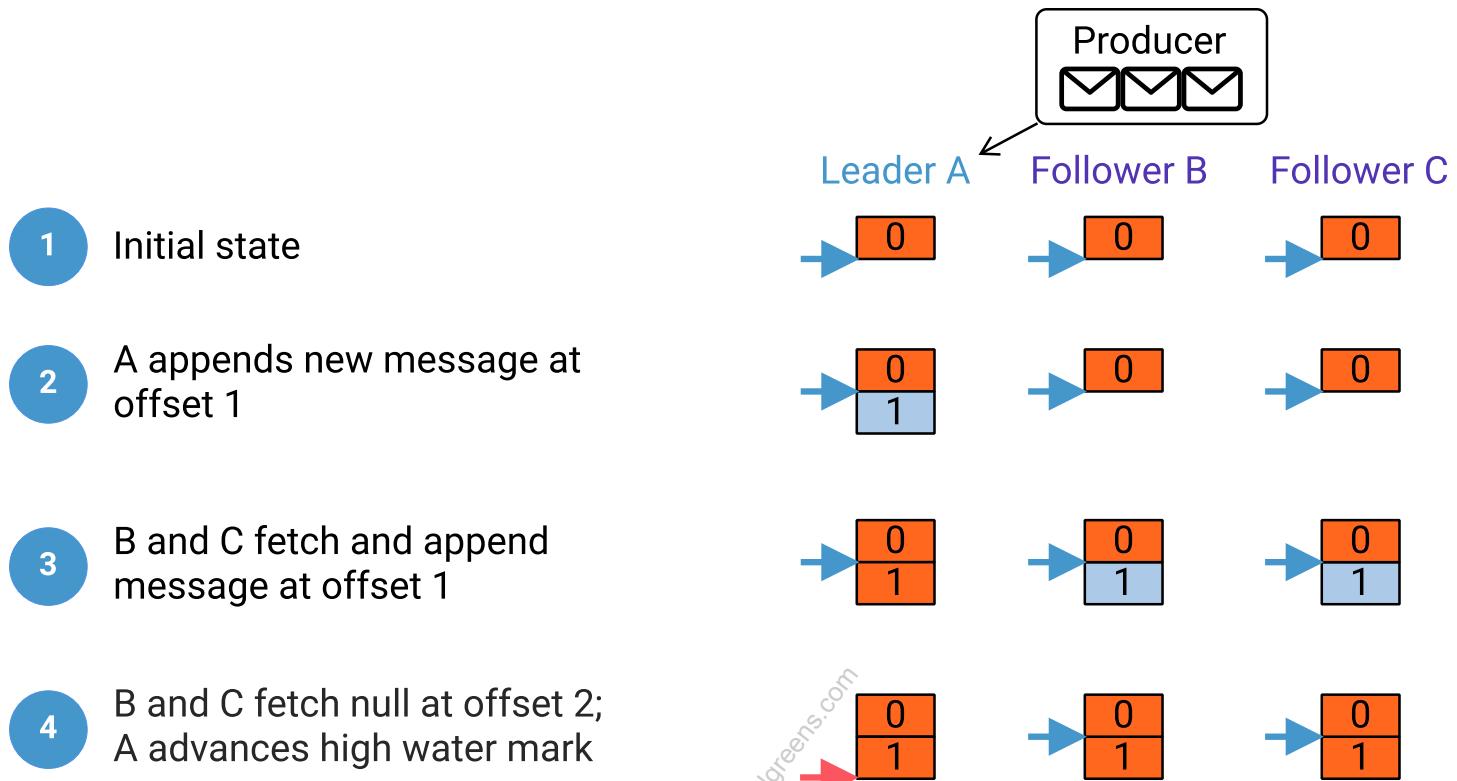
# Committing Messages with Replicas (3)



The followers on Brokers B and C independently request any available offsets from the leader. Each copies the message to its local commit log.

Now, all the replicas have the message but the leader does not know for sure that the replication has been successful. It only knows that the data was requested - there is still a chance that one of the replicas would have to retry.

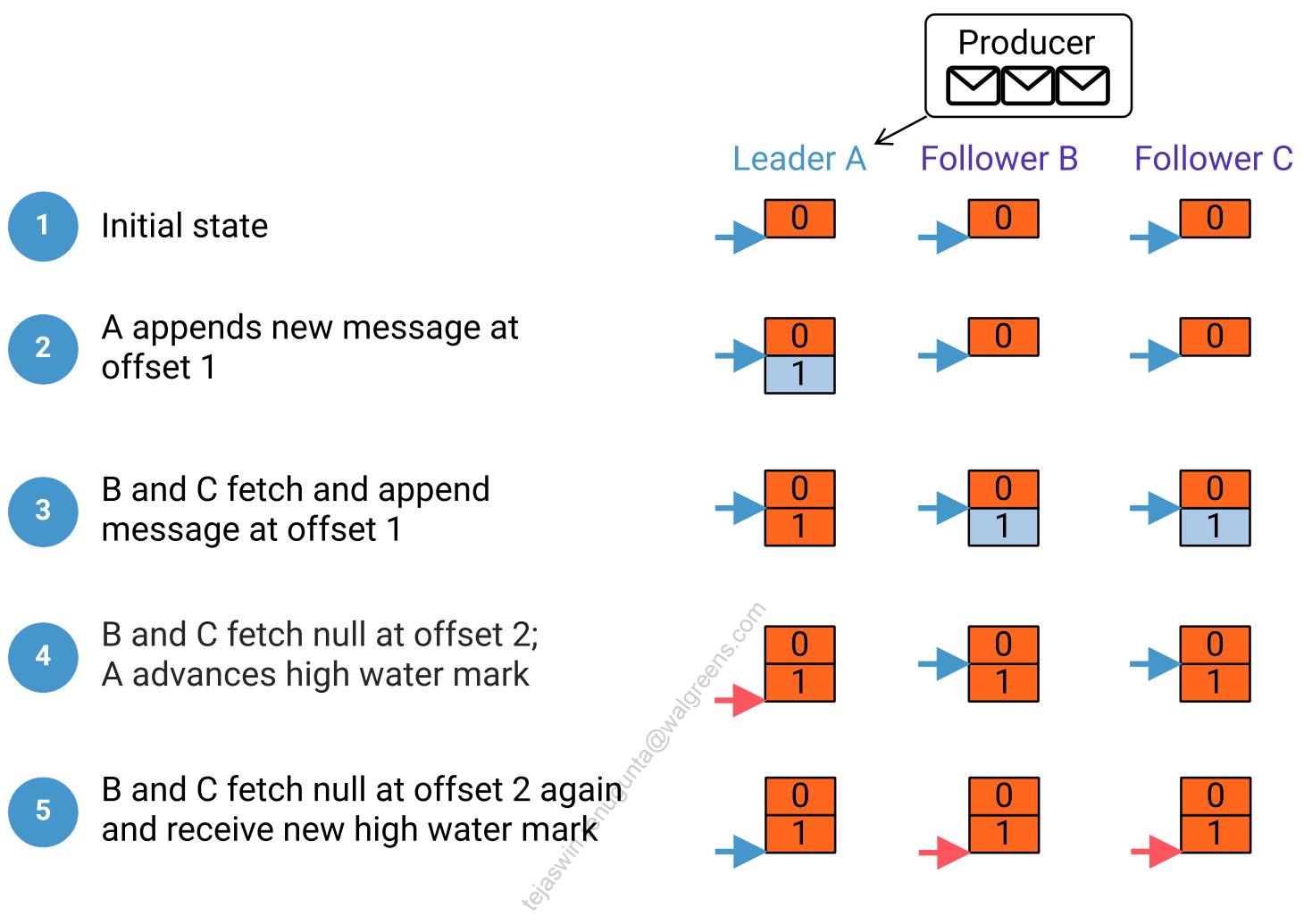
# Committing Messages with Replicas (4)



Rather than wait for a separate `ack` message, the leader is waiting for the replicas to request the next offset *past* the one that needs to be committed. Replicas will only ask for an offset if they have successfully copied the previous one. In this example, the leader knows that a follower has the message at offset 1 when it requests offset 2. Once all the replicas have requested the next offset, the leader marks the message as committed and advances the high water mark.

However, when does it send the updated high water mark to the followers? Just as with the `ack` responses, Kafka does not want to send any unnecessary messages which will decrease usable bandwidth.

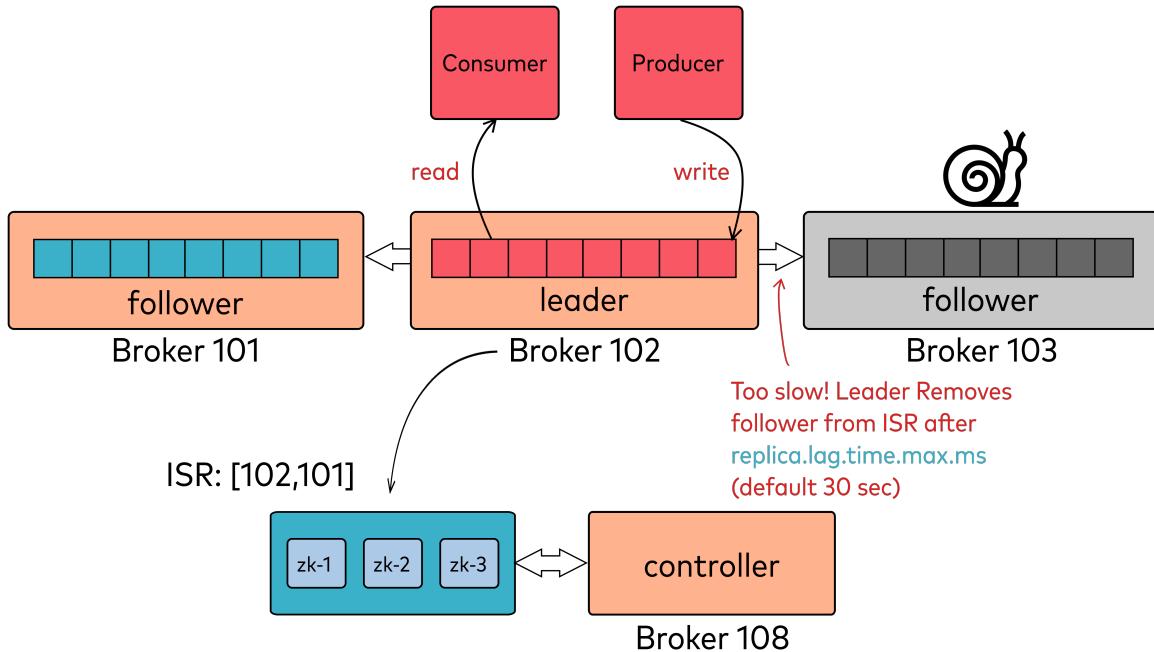
# Committing Messages with Replicas (5)



Followers constantly request more data from the leader. If new data is available (i.e., data has been added to offsets they haven't replicated yet), the leader will send the messages to the followers when the followers request it. If no data is available, the request will timeout after 500ms (default) and the followers will request data again.

Rather than send the updated high water mark to the followers as a special message, the leader will include the high water mark updates the next time the followers request new data.

# Detecting Slow Replicas



- Broker setting: `replica.lag.time.max.ms` (Default: 30 sec)
  - Leader drops slow follower from ISR list
  - Too large → slow replicas will slow down time to commit a write
  - Too small → replicas drop in and out of ISR
- Leader sends ISR changes to ZooKeeper
  - Controller listens for changes and facilitates leader elections if necessary
- When replica catches back up, it will be added back to the ISR

Recall that the condition for data to be marked committed (which makes it visible to consumers) is for all replicas in the ISR (rather than all possible replicas) to have received the message. If a replica has not requested data from the leader in `replica.lag.time.max.ms` (30 seconds by default), the replica is dropped from the ISR so that commits can happen without the delay caused by the slow replica. Once the replica catches back up to the members of the ISR list, it can be added back and its state can be used in consideration of commits again. For low latency applications, you can adjust `replica.lag.time.max.ms` down...but not too low or else you risk unnecessarily dropping members from the ISR.

# Monitoring ISR

- Monitor under-replicated Partitions with the JMX metric:
    - `kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions`
    - Alert if the value is greater than `0` for a long time
  - Track changes of ISR lists (shrinks and adds) with these JMX metrics:
    - `kafka.server:type=ReplicaManager, name=IsrExpandsPerSec`
    - `kafka.server:type=ReplicaManager, name=IsrShrinksPerSec`
- 

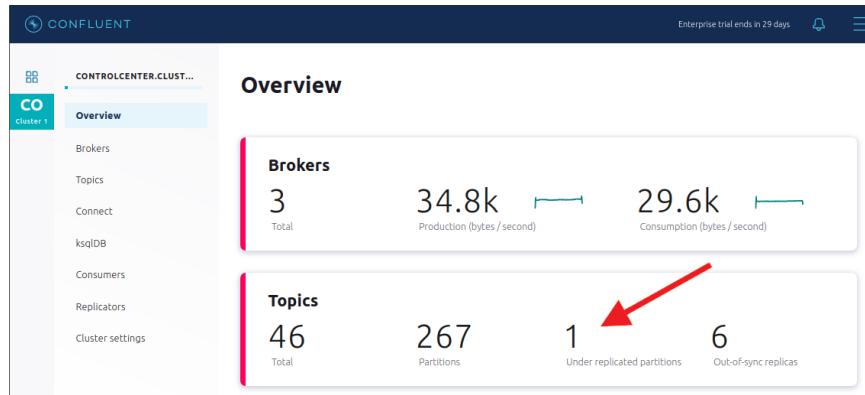
These metrics are the primary indicators of a problem within the cluster.

The `UnderReplicatedPartitions` metrics indicates the number of replicated Partitions that do not have a fully populated ISR list. A cluster should not be allowed to run indefinitely with under-replicated Partitions - another failure could result in data loss or Partitions unavailability.

The `IsrExpandsPerSec` and `IsrShrinksPerSec` metrics track changes to the ISR list. These metrics should change rarely: in a healthy cluster, all replicas should be In-Sync with the leader.

# Monitoring for Under Replicated Partitions

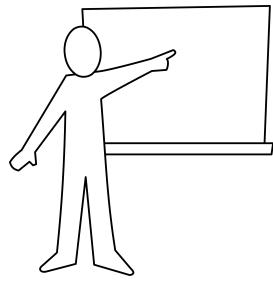
- If a Broker goes down, ISR for some of the Partitions will shrink
- Confluent Control Center shows Partition health at a glance:



The JMX metrics discussed in the previous slide can also be tracked in Confluent Control Center. The sample shown displays the counters for Under replicated and Offline Partitions in the cluster.

In Confluent Control Center, you can trigger alarms based on under-replicated and offline Partitions.

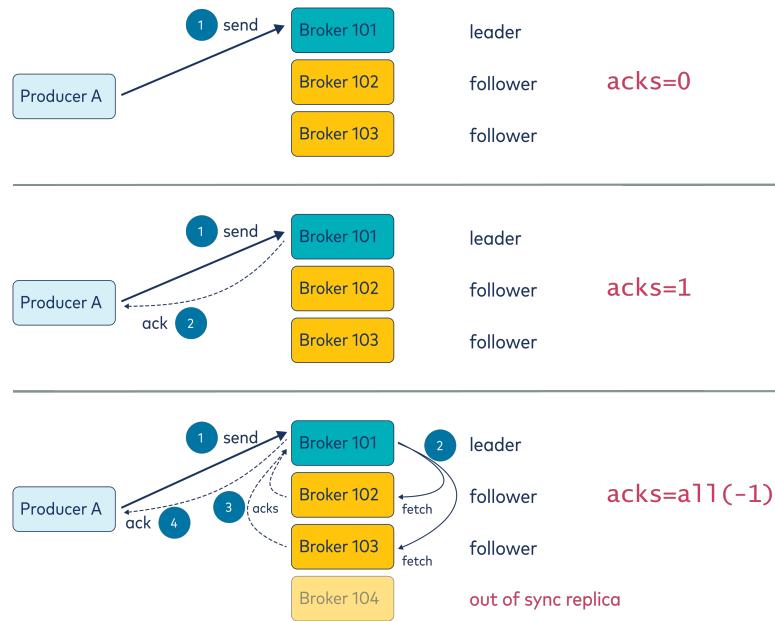
# Module Map



- Replication Review
- Advanced Replication Concepts
- Writing Data Reliably ... ←
- Broker Shutdowns and Failures
- The Kafka Log Files
- Offset Management
- Exactly Once Semantics (EOS)
- 🔧 Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

# Producer Acknowledgements



The `acks` setting is a property set on a Producer used to define when a request is successful. For `acks=all`, a request is only successful if all members of the ISR have replicated the messages in the request.

In this example, the leader is on Broker 1. As described earlier, the Producer sends messages to Broker 1 because it is the leader of the replicated Partition. Once the message is written to the leader, the followers can request data to replicate locally. The leader marks the message committed once all replicas in the ISR have the data.

The Producer can choose to receive acknowledgements to know if the write happened successfully.

- **acks=1** (default): Send ack when the message has been received by the leader. If leader fails after it sends the ack to the producer but before followers replicate, the message could be lost.
- **acks=all**: Send ack when the message is committed. This level guarantees that the data will survive even if you lose the leader. However, this level has longer latency and could see repeated data in certain failure cases. For that reason, this level is described as "at least once" delivery.



`acks=all` and `acks=-1` are equivalent

- **acks=0**: No ack required. This level is not used very often. It is primarily used when the

buffering and resending is less desirable than the chance of the occasional lost message. The stated guarantee for this level is "at most once" delivery.

As of AK 0.11.0 (CP 3.3), the `acks=all` can be combined with additional settings to provide Exactly Once Semantics (EOS). This feature set performs message deduplication at the Broker level to prevent unique messages from being added to the Partition more than once. This will be covered in detail later in the chapter.

tejaswin.renugunta@walgreens.com

# Producer Retries

Property	Description	Default
<code>retries</code>	Number times to retry sending message	<code>MAX_INT</code>
<code>retry.backoff.ms</code>	Pause added between retries	100
<code>request.timeout.ms</code>	Maximum amount of time client will wait for a response	30000
<code>delivery.timeout.ms</code>	Upper bound on the time to report success or failure after <code>send()</code>	120000



Leave `retries` at `MAX_INT`. Control retry behavior with `delivery.timeout.ms` instead.

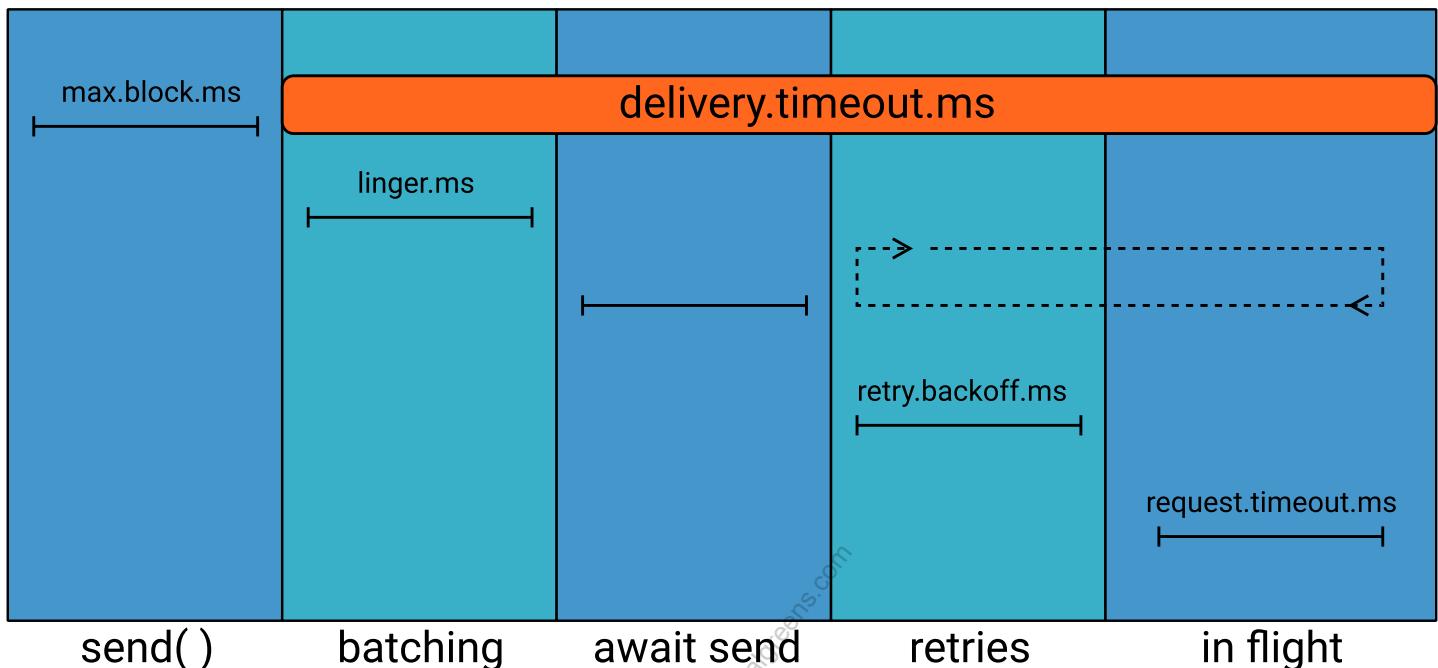
Retries are critical in any environment to make up for transient errors in the network or on the systems. Our recommendation is to leave the default value of `retries` as `MAX_INT` and to instead use `delivery.timeout.ms` to control how long Producers will retry. The `delivery.timeout.ms` property is discussed in detail in the next slide.



The `retries` property doesn't make sense if `acks=0`

# Producer's Delivery Timeout

- Producer property `delivery.timeout.ms` puts an upper bound on the time to report success or failure after a call to `send()` returns.



The diagram illustrates the lifecycle of a batch of records created by a Producer. All properties shown are configurable in the Producer code. The main purpose of this slide is that the `delivery.timeout.ms` property on the Producer allows us to put an upper bound on the time to report success or failure after a call to `send()` returns.

- The `max.block.ms` property controls how long `KafkaProducer.send()` and `KafkaProducer.partitionsFor()` will block. Blocking could happen when the buffer is full or if metadata is unavailable.
- The `linger.ms` property is the maximum amount of time that the Producer will take to accumulate records into a `RecordBatch`.

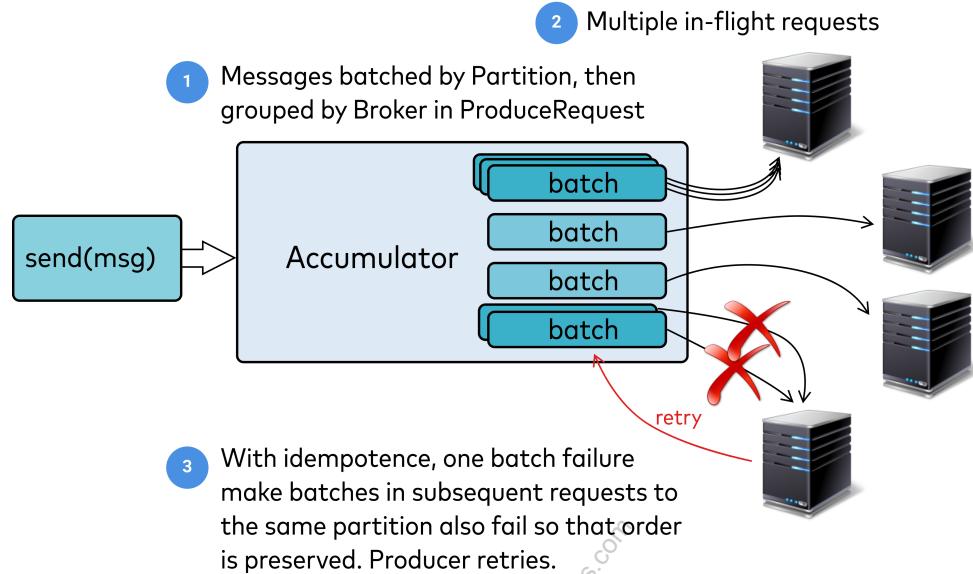


Batching will be explored in more detail in an upcoming module and lab exercise.

For more details on delivery timeout, see KIP 91 at  
<https://cwiki.apache.org/confluence/display/KAFKA/KIP-91+Provide+Intuitive+User+Timeouts+in+The+Producer>

# Preserve Message Send Order

- Retries with multiple in-flight requests can cause order to change
- To guarantee message order, set `enable.idempotence` to `true` on the Producer



There is a Producer property called `max.in.flight.requests.per.connection` that defaults to 5, which means that you can have up to five unacknowledged ProduceRequests from a single producer to a Broker at any given time. In other words, a Producer can send a ProduceRequest to a Broker and then send up to four more before the first is acknowledged. This is great for throughput but can result in out of order delivery.

For example, assume the Producer sends message batch m1 and then message batch m2. If m1 fails but m2 succeeds, m2 could be added to the commit log before m1.

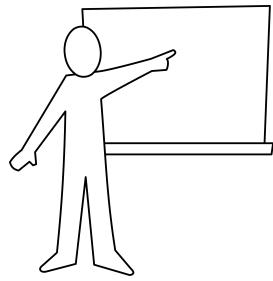
To get around this, enable idempotent Producers. Each message batch has a "sequence ID" that tells us the order the messages were produced. When idempotence is enabled, batches of messages are ordered by sequence ID in priority queues, so order will be preserved even on a retry. In the illustration, we see that when a message batch fails, all subsequent message batches to the same partition will also fail with an `OutOfOrderSequenceException`. Batches will sit on the Producer for a time configured by `delivery.timeout.ms` (Default: 2 minutes) to allow time to recover. Enabling idempotence has negligible overhead, so it is highly recommended in most circumstances.



There are a couple of caveats when enabling idempotent Producers: the value of `max.in.flight.requests.per.connection` must be 5 or less, the `acks` setting must be set to `all`, and `retries` must be greater than 0. These are reasonable caveats in any situation where message order guarantees matter. Students will learn more about idempotent Producers later in the module. For more information, see <https://issues.apache.org/jira/browse/KAFKA-5494>

tejaswin.renugunta@walgreens.com

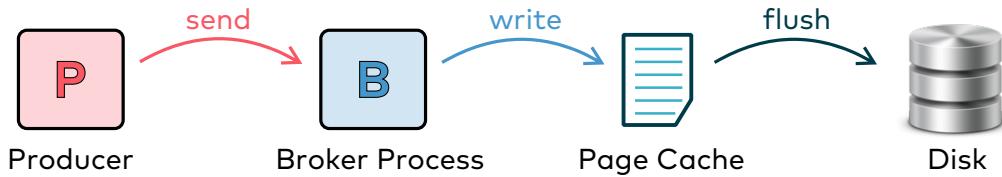
# Module Map



- Replication Review
- Advanced Replication Concepts
- Writing Data Reliably
- Broker Shutdowns and Failures ... 
- The Kafka Log Files
- Offset Management
- Exactly Once Semantics (EOS)
-  Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

# Page Cache and Flushing to Disk



- Messages are written to Partitions
- Partitions are made up of **log segment files** (new segment every 1 GB by default)
- Log segments are written to the in-memory **page cache** for performance
  - Kafka client fetch requests benefit from **zero-copy transfer**
- Page cache is **flushed to disk** when:
  - Brokers have a clean shutdown
  - OS background "flusher threads" run

The data format of messages saved into the log files is exactly the same as what the Broker receives from the Producer and sends to its Consumers, which allows for zero-copy transfer. Zero-copy transfer is when data flows directly between page cache and network buffer without first being copied to userspace. This allows for excellent Consumer throughput.

The implication of this slide is that if the Broker's machine fails before the OS has flushed data to disk, that data will be lost. If the Topic is replicated, then when the Broker comes back online, the data will be recovered from the leader replica. Without replication, there may be permanent data loss.

Kafka does have its own flush policy. It can be set to trigger flushing (`fsync`) by either the number of messages (`log.flush.interval.messages`) or time (`log.flush.interval.ms`) since the last flush. However, those settings default to infinite (essentially disabling `fsync`) because Kafka prefers to allow the operating system background flush capabilities (i.e. `pdflush`) as it is more efficient. We highly recommend keeping these settings at default.

When users look at the `*.log` files, it shows data that is both flushed to disk and still in the page cache (OS buffer) that has not yet been flushed. There are linux tools (e.g. `vmtouch`) that show what has and hasn't been flushed.

# Maintaining the List of In-Sync Replicas

- A message is considered committed if it is received by every replica in the ISR list
- Leader persists changes to ISR list in ZooKeeper, which is monitored by the cluster's Controller
  - If a follower fails:
    - It is dropped from the ISR list by the leader
    - Leader commits using the new ISRs
  - If a leader fails:
    - Controller elects new leader from followers
    - Controller pushes new leader and ISR to ZooKeeper first and then to Brokers for local caching

At a high level, all the followers that are caught up with the leader are ISRs. Recall: to commit a message, the leader must wait for the message to be received by all ISRs. You don't want to wait forever if a follower fails, so failed followers will be removed from the ISR list. This allows the leader to commit new messages with fewer replicas, but the Partition is now considered under-replicated.

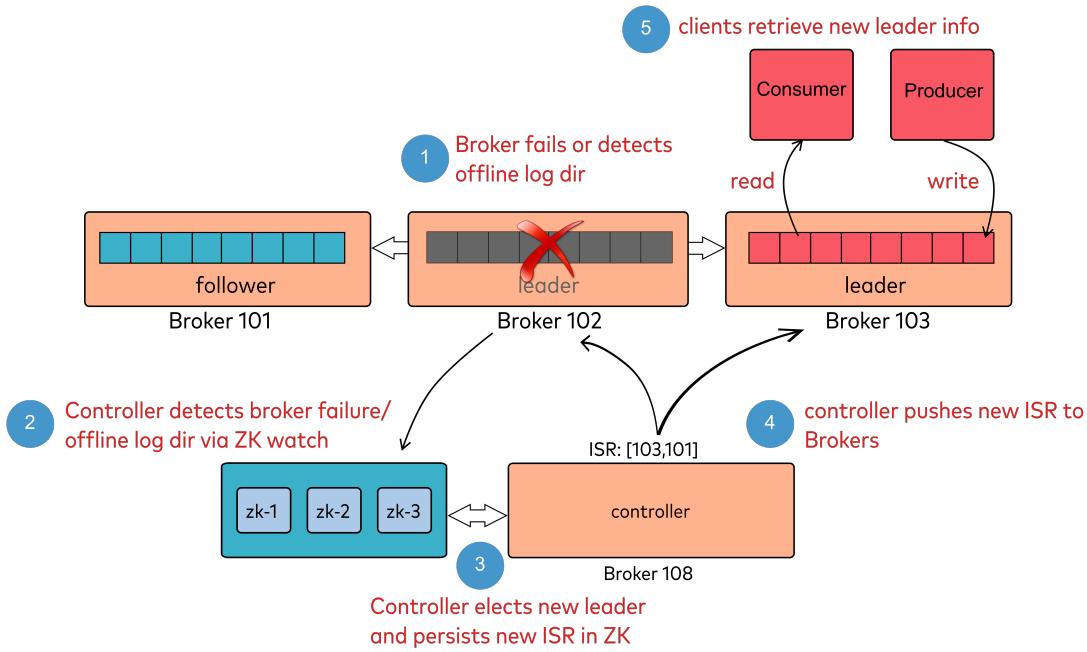
If a leader fails, another replica must be elected as a new leader. To do this, we must be able to reliably store and maintain the ISR list. Kafka achieves this by storing the ISR, leader information, and other metadata in ZooKeeper. The Controller is in charge of electing a new leader from the ISR and updating replica information across the cluster (updating the metadata in ZooKeeper and sending it to the caches of all Brokers).

How many replicas can fail? By default, Kafka allows a Partition to run with a single replica. If you have n replicas, you can tolerate n-1 failures. However, there is a Topic configuration called `min.insync.replicas` that will be discussed later in this chapter. That setting allows you to specify a minimum number of replicas for the Partition to function normally.



When lead Brokers fail, it can lead to lopsided leadership where some Brokers are leaders for more Partitions than others. Kafka will automatically remedy this so long as the Broker property `auto.leader.rebalance.enable` is configured to `true`. This rebalance will happen with a frequency set by `leader.imbalance.check.interval.seconds` (Default: 5 minutes) and affect Brokers whose percentage of non-preferred leader replicas is greater than an amount set by `leader.imbalance.per.broker.percentage` (Default 10%).

# Example of Leader Failure



If the leader of a Partition fails, its ZooKeeper session times out. The Controller detects this timeout and elects a new leader from the ISR list. This information will be recorded in ZooKeeper (to protect against Controller failure) and then propagated to all the Brokers for client metadata updates.

Both Producers and Consumers need to know which Broker is the leader for the Partition they need to contact. This is done through a metadata request. When the client starts, it does not know which Brokers are the leaders of which Partitions so it issues a metadata request to a Broker (any Broker). This is why the Controller caches the Partition information on all the Brokers - so any Broker can respond to a metadata request. After receiving the updated metadata, the client will know who the leader is for each Partition and will communicate with the appropriate Brokers. If a Broker fails, the client will get an error and will refresh its metadata.

# Partitions Without Leaders

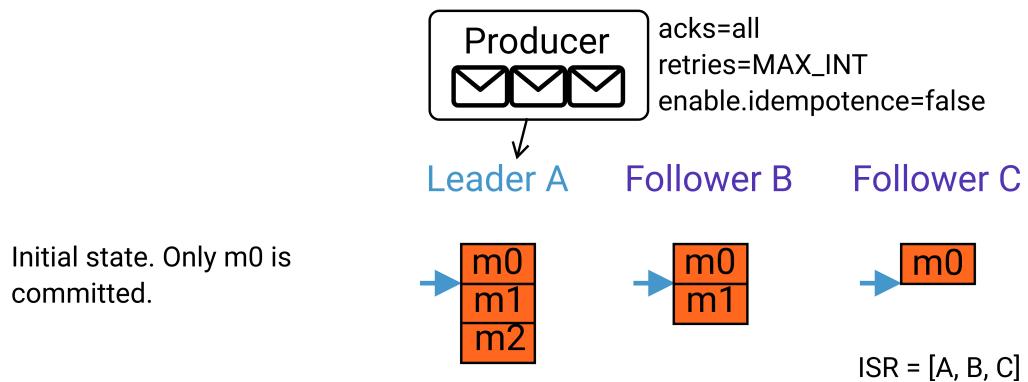
- Leader failure makes Partition unavailable until re-election
    - Producer `send()` will retry according to `retries` configuration
    - Callback raises `NetworkException` if `retries == 0`
  - Track Offline Partitions with JMX metric:
    - `kafka.controller:type=KafkaController,Name=OfflinePartitionsCount`
- 

The `OfflinePartitionCount` metric shows the number of Partitions that do not have an active leader. These Partitions can perform no reads nor writes until a new leader is elected.

During the transient period when a leader Broker fails, Producers should expect error messages.

- If `retries == 0`
  - The callback will receive an exception with the following message "org.apache.kafka.common.errors.NetworkException: The server disconnected before a response was received."
  - Message will not be written to Kafka.
- If `retries > 0`
  - No callback but log message will be written "Sender:298 - Got error produce response with correlation id 5 on Topic-Partition hello\_world\_topic-0-0, retrying (0 attempts left). Error: NETWORK\_EXCEPTION"
  - Message will be written to Kafka via new leader on a subsequent retry.

# Example of Replica Recovery (1)



For this example, assume a single Partition with three replicas. The Leader is on Broker A. Initially, all replicas are In-Sync. The Producer has sent three messages (m0, m1, m2). Message m0 has been replicated across the ISR list and is committed. Broker B has replicated m1, but Broker C has not replicated any messages other than m0.

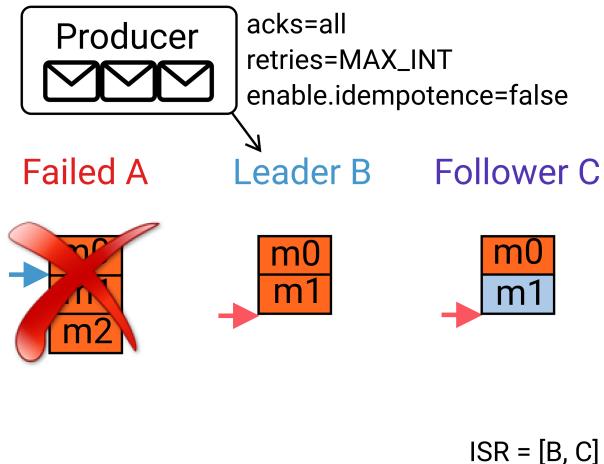
# Example of Replica Recovery (2)

A fails before committing m1 and m2.

B becomes leader.  
Leader epoch increments from 0 to 1.

C follows B and fetches m1.  
B advances its high water mark.

C fetches again and advances its high water mark.



When Broker A fails, a new leader must be elected. The Controller can select Broker B or Broker C since both are In-Sync. Assume Broker B becomes the new leader. The leader epoch is updated to the log end offset of the new leader, which for Broker B is the second offset (the offset containing m1). Broker B knows to retain m1 since it was received from the most recent leader in the `leader-epoch-checkpoint` file. When Broker C fetches m1 from its new leader, m1 is considered committed and the high watermark advances.

If the Producer uses `acks=all`, only message m0 will be acknowledged as committed. Since Broker A failed before the message was committed, the message was not acknowledged to the Producer. From the Producer's perspective, if you get an error (or a timeout) you must resend, but that could produce duplicates.

Broker A was never able to `ack` the Producer for m1, so the Producer will resend m1.



If Broker C had become the new leader, m1 would not have been committed since the current leader never received it. The offset for that message will be overwritten on Broker B when it starts replicating off of its new leader, Broker C.

Why wasn't message one lost?:

- Leaders do not truncate
- Followers truncate and re-fetch messages from the leader using the leader epoch (this will be shown in an upcoming slide)

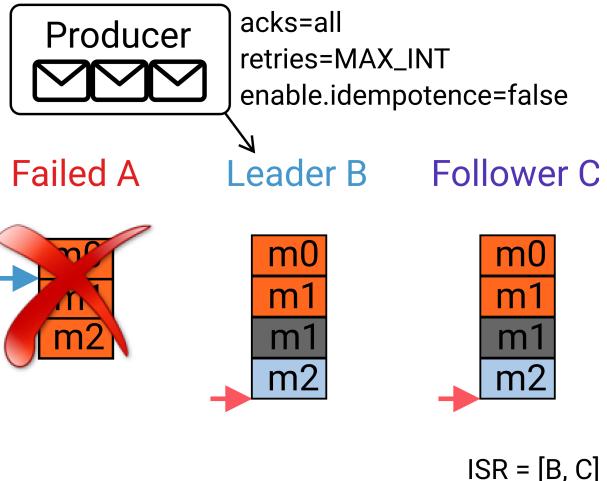
# Example of Replica Recovery (3)

Producer never received acks for m1 or m2, so it retries to B.

Idempotence is disabled, so m1 is duplicated.

C fetches once to get m1 (dup) and m2.

C fetches again to advance high water mark.



If the Producer was set with `acks=all`, this scenario results in repeated data. Messages 1 and 2 were never marked committed by the original leader that the Producer was sending to (Broker A), so that Broker never acknowledged those messages. The new leader (Broker B) did commit m1, but since it was not the original recipient, it did not know who to send the acknowledgement to. As a result, m1 and m2 timed out waiting for acknowledgement and the Producer retried both. In this case, m1 was received twice to illustrate the "at least once" delivery guarantee of `acks=all`.

What would have happened if the Producer had used `acks=1`? In that case, all messages (m0, m1, m2) would have been acknowledged by the original leader (Broker A). The Producer would have considered those writes complete and would not retry after the failure of Broker A. This means that m2 would be lost if Broker B became the new leader; both m1 and m2 would be lost if Broker C became the leader. This illustrates the "at most once" delivery guarantee of `acks=1`.

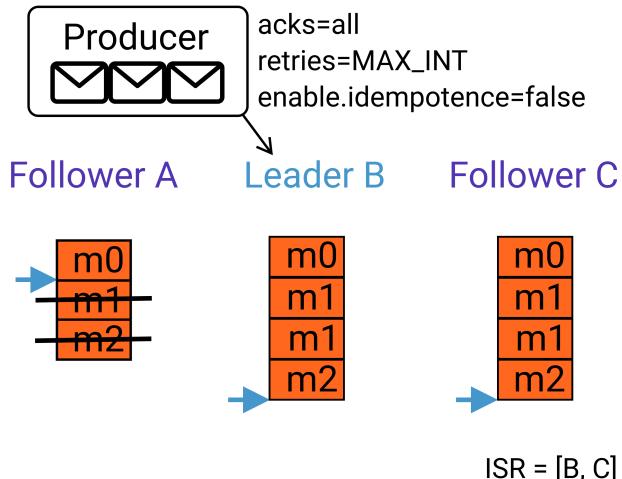
# Example of Replica Recovery (4)

A restarts and follows B.

A receives metadata from Controller.

A fetches leader epoch from leader B.

A truncates to place where leadership changed.



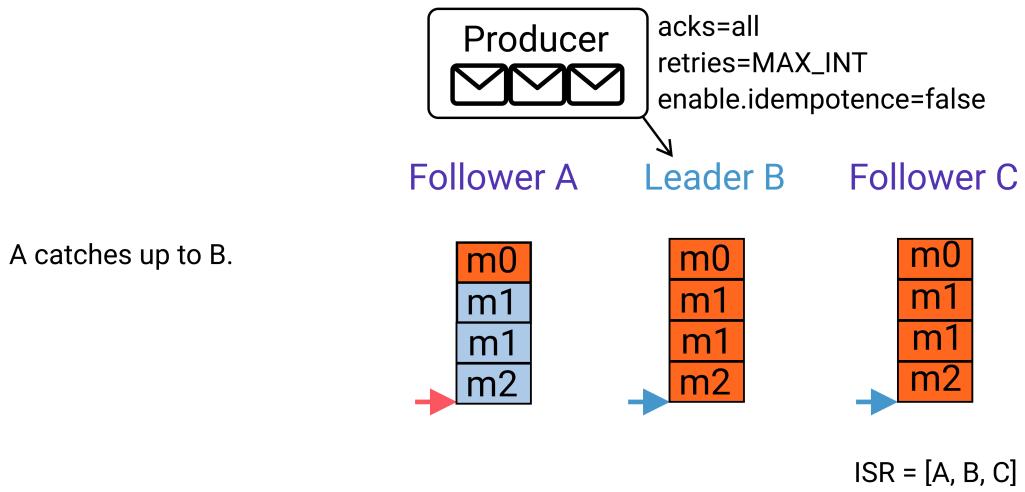
When a Broker restarts, it needs to catch up to the current state of the leader. The recovering Broker must ensure its log is identical to the log of the current leader. In particular, there may be some messages in this replica's log that were never committed and need to be cleaned up. The restarted Broker connects to ZooKeeper, which alerts the Controller to send it Partition metadata. The restarted Broker can then request the leader epoch from current leader so that replicating will start from the last offset where the leader of the Partition changed. Anything past that location is considered untrusted and will be truncated.

Prior to Kafka 0.11, Broker A would have truncated to the last known high water mark saved to the `replication-offset-checkpoint` file. This method could lead to rare cases where logs could diverge or committed messages could be lost. For more information, see [KIP 101](#).

For more details about how exactly the Controller handles Broker recovery, see the source code at

<https://github.com/apache/kafka/blob/trunk/core/src/main/scala/kafka/controller/KafkaController.scala>. Especially note the `onBrokerStartup` function.

# Example of Replica Recovery (5)



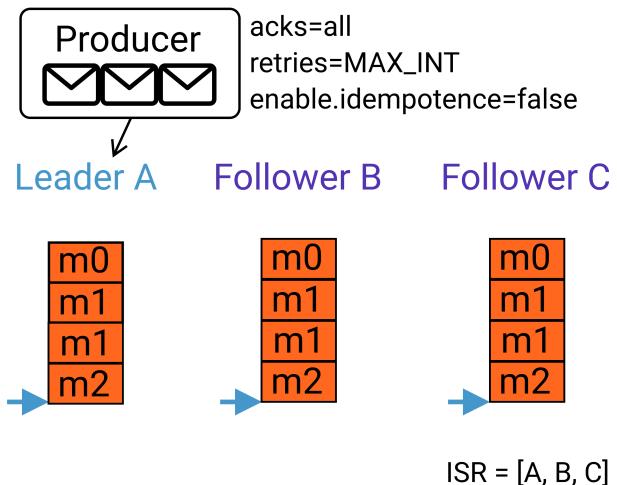
Starting from its own **leader-epoch-checkpoint**, the recovering replica will copy the offsets from the leader in order until its high water mark matches that of the leader. Once that happens, the Broker can be added back to the ISR list.

# Example of Replica Recovery (6)

After a while, Controller notices imbalanced leadership and re-elects the preferred replica A.

Leader epoch increments from 1 to 2.

B and C now follow A.



The restored replica does not immediately regain its role as leader. Leader elections are disruptive, so Kafka only does non-failure re-elections on a scheduled interval (Default: 5 min) according to the property `leader.imbalance.check.interval.seconds` and only if the percentage of non-preferred leaders on a Broker is greater than a certain amount (Default: 10%) determined by the property `leader.imbalance.per.broker.percentage`. Balancing lead replicas is the Controller's job (with state persisted in ZooKeeper), which is why the ISR is able to be restored with Broker A holding the preferred replica.

# Tradeoff: Availability vs Durability

- Topic configuration property: `unclean.leader.election.enable`
  - Determines whether a new leader can be elected even if it is not In-Sync, if there is no other choice
  - Can result in data loss if enabled (Default: `false`)
  - Monitor with the JMX metric:  
`kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec`
- Topic configuration property: `min.insync.replicas` (Default: 1)
  - Producer receives a `NotEnoughReplicas` exception if too few ISRs
  - Stronger guarantees when used with `acks=all` on the Producer

---

Both of these properties are set at a Topic level, and cluster-wide defaults can be set at the Broker level.

`unclean.leader.election.enable`: By default, a leader is selected from the ISR list. This makes the most sense because that guarantees that the data is consistent up to the high water mark. But what if the leader fails and the only available replicas are out of sync? If the config is set to `true`, the Partition is available for writes immediately but will lose any committed messages it had not synchronized before the leader failed. If the config is set to `false`, the Partition will be offline until one of the In-Sync replicas come back. Default behavior is `false` because of its stronger durability guarantee.

`min.insync.replicas`: By default, Kafka allows the ISR list to drop to one member—just the leader. However, this does not meet the data redundancy requirement for many big data environments. It is common to configure a Topic with replication factor of 3 and `min.insync.replicas` of 2. In this example, as long as the number of members of the ISR list is not less than `min.insync.replicas`, the Partitions behave normally. However, if a Partition drops below that setting, the Partition will return exceptions for any produce requests. This guarantees a minimum replication level for any data accepted into the Topic. Consume requests are unaffected.

# Discussion

- What configuration settings are important if you want to minimize data loss?
  - What if you wanted to also guarantee correct message ordering?
  - How would you change these settings if you were less concerned about data loss and more concerned with lowering end-to-end latency?
- 

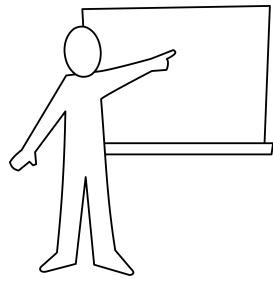
Consider giving students a few minutes to gather their thoughts and make notes. Then invite students to share their ideas with a peer. Finally, select students to share their ideas and the ideas of their peers in a whole-class discussion. Remind students that first-draft ideas are always a welcome starting point. Record ideas as they arise for all to see. Make sure to elaborate whether each configuration is set on a Producer, a Topic, or a Broker.

- What configuration settings are important if you want to minimize data loss?
  - Topic: `replication.factor` should be greater than 2.
  - Producer: `acks` should be set to `all`
  - Topic: `min.insync.replicas` should be greater than 1
- What if you wanted to also guarantee correct message ordering?
  - Producer: `enable.idempotence` should be set to `true`
- How would you change these settings if you were less concerned about data loss and more concerned with lowering end-to-end latency?
  - We could set `unclean.leader.election.enable` to `true` for the Topic
  - We could make sure `max.in.flight.requests.per.connection` is greater than 1

It might be interesting to share the following performance analysis of `acks` and `max.in.flight.requests.per.connection` published in August 2017.

<https://cwiki.apache.org/confluence/display/KAFKA/An+analysis+of+the+impact+of+max.in.flight.requests.per.connection+and+acks+on+Producer+performance>

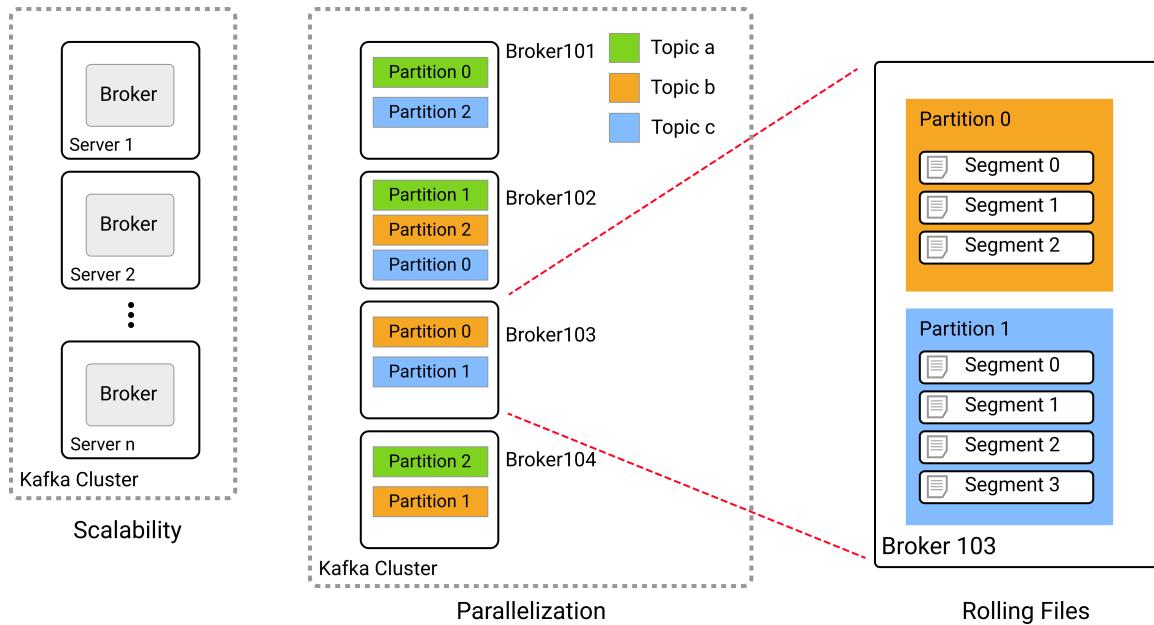
# Module Map



- Replication Review
- Advanced Replication Concepts
- Writing Data Reliably
- Broker Shutdowns and Failures
- The Kafka Log Files ... ←
- Offset Management
- Exactly Once Semantics (EOS)
-  Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

# Log Files and Segments



Recall that Brokers hold Partitions, and Partitions are made up of log segment files. The term **roll** means that the active segment file is closed and a new file is opened.

# Log File Subdirectories

- Kafka **log segment** files are sometimes called data files
- Each Broker has one or more data directories specified in the `server.properties` file, e.g.:

```
log.dirs=/var/lib/kafka/data-a,/var/lib/kafka/data-b,/var/lib/kafka/data-c
```

- Each Topic-Partition has a separate subdirectory
  - e.g., `/var/lib/kafka/data-a/my_topic-0` for Partition 0 of Topic `my_topic`
- Brokers detect log directory failures and notify Controller

---

A best practice is to mount file systems for each directory in `log.dirs` since they will quickly fill with data. Having a dedicated disk per log directory also means that a disk failure will not take the Broker completely offline.

When specifying multiple paths in `log.dirs`, Kafka assigns Partitions across the directories in a round-robin fashion. Partitions can be re-assigned to specific log directories using the `kafka-reassign-partitions` tool, which is mentioned later in the course.



When a Broker detects a log directory failure, it doesn't notify the Controller directly. The protocol is described in more detail here:  
<https://cwiki.apache.org/confluence/display/KAFKA/KIP-112%3A+Handle+disk+failure+for+JBOD>

# File Types Per Topic Partition

- Per log segment

.log	Log segment file holds the messages and metadata
.index	Index file that maps message offsets to their byte position in the log file
.timeindex	Time-based index file that maps message timestamps to their offset number
.snapshot	If using idempotent Producers, checkpoints PID and seq #
.txnidex	If using transactional Producers, indexes aborted transactions

- Per partition

leader-epoch-checkpoint	Maps the leader epoch to its corresponding start offset
-------------------------	---

---

The data format of messages saved into the log files is exactly the same as what the Broker receives from the Producer and sends to its Consumers.

The `*.index` file is not continuous like the `*.log` file—there may be jumps. There is an interval (`index.interval.bytes`) that controls how frequently Kafka adds an index entry to its offset index (default is 4k bytes). More frequent indexing allows reads to jump closer to the exact position in the log but makes the index larger.

The `*.timeindex` index file enables timestamp-based functions, such as:

- Searching message by timestamp. This is useful to rewind offsets if applications need to re-consume messages for a certain period of time, or in a multi-datacenter environment because the offset between two different Kafka clusters are independent and users cannot use the offsets from the failed datacenter to consume from the DR datacenter. In this case, searching by timestamp will help because the messages should have same timestamp if users are using the Producer option `CreateTime`.
- Time-based log rolling and log retention.

The `leader-epoch-checkpoint` and `.snapshot` files were added in 0.11.0. The `.snapshot` file will be discussed in further detail in an upcoming section on EOS.

# Example of Log Files

Example of one Broker's subdirectory for Topic `my_topic` with Partition `0`

```
$ ls /var/lib/kafka/data-b/my_topic-0
000000000000283423.index
000000000000283423.timeindex
000000000000283423.log
...
0000000000008296402.index
0000000000008296402.timeindex
0000000000008296402.log
leader-epoch-checkpoint
```

Segment files are named for the first offset tracked by that set of files. In the example, the files named `000000000000283423.*` manage messages in offsets from 000000000000283423 to 0000000000008296401 ([the name of the next segment file] - 1).

If using idempotent Producers, Partitions may now also have one or more `.snapshot` files. This will be discussed in more detail when discussing EOS.

tejaswin.renugunta@wistarens.com

# Log Segment Properties

- A Partition is comprised of one or more **segments**
  - Each `.log` data file is a segment of the overall Partition
- Each `.log` filename is equal to the offset of the first message it contains:
  - `00000000000049288237.log`
- The Partition **rolls** to a new segment file if any are exceeded:
  - `log.segment.bytes` (Default: 1GB)
  - `log.roll.ms` (Default: 168 hours)
  - `log.index.size.max.bytes` (Default: 10MB)
- Separately control when to roll the segment file for the offsets Topic
  - `offsets.topic.segment.bytes`

All of the settings on this page can be set as cluster-wide defaults or at the Topic level. Topic level changes will override the defaults.

The purpose of segment files (instead of a single monolithic file) is to provide granular control of the data when purging old data.

Most environments will use the `log.segment.bytes` as the roll threshold, but sometimes setting `log.index.size.max.bytes` may be appropriate. The property `log.index.size.max.bytes` describes the maximum size of the `.index` file that indexes offsets contained in the associated `.log` file. It is most useful to tune this in environments where message sizes are very small and the standard log segment size threshold would not provide enough granular control over log rolling due to the sheer number of messages in each segment file.

Note that there is a separate property (`offsets.topic.segment.bytes`) to manage the roll behavior of the `consumer_offsets` Topic. This mission-critical Topic is so important that rather than risk affecting its behavior when standard Topic configurations are changed, `consumer_offsets` has a separate set of configuration properties.

# Checkpoint Files

In addition, each Broker has two checkpoint files:

- **replication-offset-checkpoint**
  - Contains the **high water mark** which is the offset of the last committed message
  - On startup, followers use this to truncate any uncommitted messages
- **recovery-point-offset-checkpoint**
  - Contains the offset up to which data has been flushed to disk
  - During recovery, Broker checks whether messages past this point have been lost

---

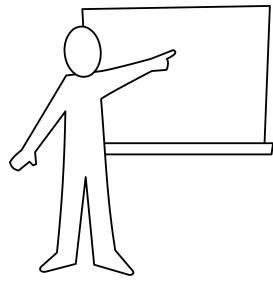
These files are located in the `log.dirs` directories, not the Partition subdirectories. They are not intended to be manually administered and should not be altered or deleted.

- The **recovery-point-offset-checkpoint** file is updated by the broker after a segment rolls and upon controlled shutdown. By default, this is the only time the broker knows for sure that records have been flushed to disk. This could be different depending on how the `log.flush.interval.messages` and `log.flush.interval.ms` properties are configured. For additional information, review the documentation.

<https://kafka.apache.org/documentation/#log.flush.interval.messages>

<https://kafka.apache.org/documentation/#log.flush.interval.ms>

# Module Map



- Replication Review
- Advanced Replication Concepts
- Writing Data Reliably
- Broker Shutdowns and Failures
- The Kafka Log Files
- Offset Management ... ↵
- Exactly Once Semantics (EOS)
- 🔧 Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

# Consumer Offset Management

- Tracks consumption per Topic Partition
    - As the consumer processes messages, it periodically commits the offset of the next message to be consumed
  - Offsets are committed to an internal Kafka Topic `__consumer_offsets`
  - Offsets can be committed automatically or manually by the Consumer
- 

Consumer Groups need to keep track of offsets to avoid rereading data after a restart. Each Consumer will track the offsets it has read from its assigned Partition(s). The Consumers need to commit (checkpoint) the offsets to the `__consumer_offsets` Topic as they read data so that the Partition can be read from where they left off in case of a failure. Consumers can be configured to commit offsets automatically (default) or manually by the application developer.

tejaswin.renugunta@walgreens.com

# Important Configuration Settings for Offsets

- `__consumer_offsets` auto-created upon first consumption
- Scalability: `offsets.topic.num.partitions` (Default: 50)
- Resiliency: `offsets.topic.replication.factor` (Default: 3)



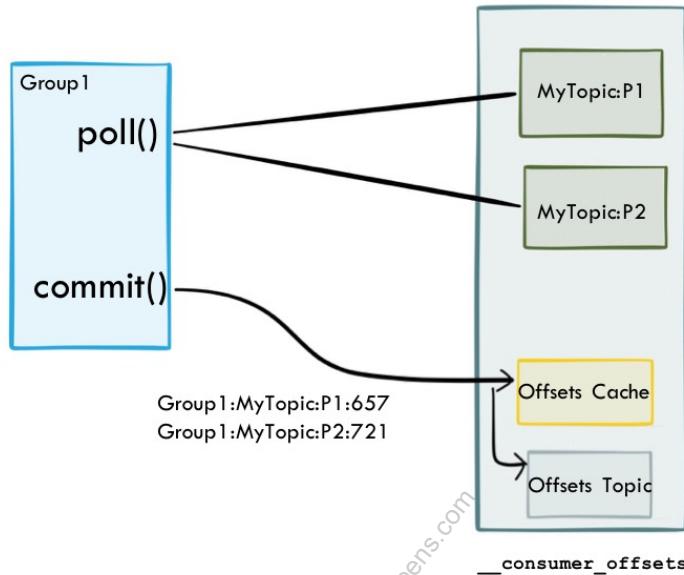
If there aren't enough Brokers, auto-creation of `__consumer_offsets` fails. Consumers should only begin consuming after all Brokers are running.

The setting `offsets.topic.replication.factor` will be enforced during auto Topic creation for the offsets Topic. If the number of Brokers is less (e.g., 2), then there will be a "GROUP\_COORDINATOR\_NOT\_AVAILABLE" error until 3 Brokers come online.

If you need to build a smaller, non-production cluster (e.g., for development), the `__consumer_offsets` Topic can be created manually.

In versions of Kafka prior to 0.11.0, the `__consumer_offsets` Topic could be automatically created for clusters with less than three Brokers. The Topic would be created with a replication factor equal to the current Broker count. However, if the cluster was expanded to three or more Brokers, the offsets Topic would generate errors due to the fact that the replication factor was below the required value of 3.

# Consumers and Offsets (Kafka Topic Storage)



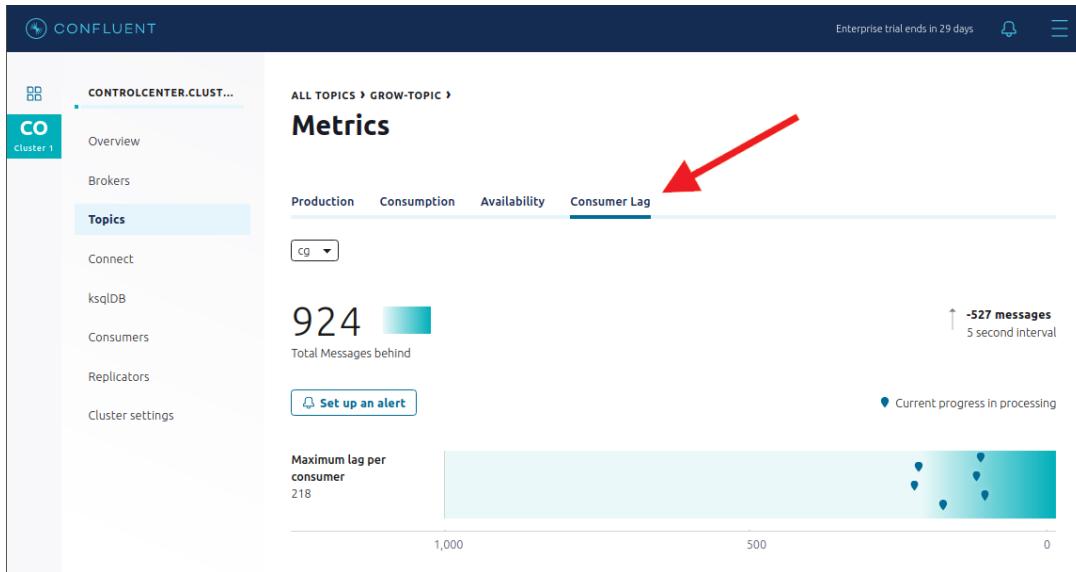
When a Consumer commits its offsets to the offsets Topic, it sends four data points:

- Consumer Group name
- Topic name
- Partition number
- Next offset to be read

Note that the Consumer name is not included. This is because Consumers within a Consumer Group are meant to be interchangeable for high availability. It should not matter which Consumer read from a specific Partition. Because the Consumer name is not part of the offsets Topic data, even single Consumers need to be part of a Consumer Group.

Consumer offset data *MUST* be read in order, else it is useless. The order of the offsets is maintained by the fact that the `__consumer_offsets` Topic uses semantic (key-based) partitioning, using the Consumer Group name as the key. This ensures all Consumer offset information for a given Consumer Group lands on the same Partition of the `__consumer_offsets` Topic.

# Checking Consumer Offsets (1)



Consumer Group lag can also be tracked in Confluent Control Center. View Consumer-Partition lag across Topics for a Consumer Group. Alert on max Consumer Group lag across all Topics.

# Checking Consumer Offsets (2)

- Look for the Current Offset and Lag

```
$ kafka-consumer-groups --group my-group \
    --describe \
    --bootstrap
--server=broker101:9092,broker102:9092,broker103:9092
```

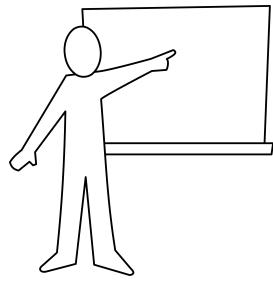
TOPIC	PARTITION	CURRENT OFFSET	LOG END OFFSET	LAG	CONSUMER-ID
my_topic	0	400	500	100	consumer-1_/127.0.0.1
my_topic	1	500	500	0	consumer-1_/127.0.0.1

The lag shown by this command depends on the offset commit interval, so it is essentially reflecting the last commit value. It is useful to make sure the members of the group are what you're expecting.

For real-time lag, the **MaxLag** JMX metric is a better tool.

tejaswin.renugunta@walgreens.com

# Module Map



- Replication Review
- Advanced Replication Concepts
- Writing Data Reliably
- Broker Shutdowns and Failures
- The Kafka Log Files
- Offset Management
- Exactly Once Semantics (EOS) ... ←
- 🔧 Hands-on Lab: **Investigating the Distributed Log**

tejaswin.renugunta@walgreens.com

# Motivation for Exactly Once Semantics (EOS)

- Write real-time, mission-critical streaming applications that require guarantees that data is processed "exactly once"
- Exactly Once Semantics (EOS) bring strong **transactional** guarantees to Kafka
  - Prevents duplicate messages from being produced by client applications (idempotent producers)
  - Ensures messages in a transaction are all consumed or none are consumed (atomic messages)
- Sample use cases:
  - tracking ad views for billing
  - processing financial transactions
  - tracking inventory in the supply chain

---

In the previous section, students explored how Kafka's offset management is used to track messages and Consumer progress. This section elaborates on the use of offsets to describe how Kafka can achieve Exactly Once Semantics (EOS). This section is a very high level overview of EOS. Feel free to direct students to these resources for a more thorough discussion of the subject:

<https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>

<https://www.confluent.io/online-talk/introducing-exactly-once-semantics-in-apache-kafka/>

Exactly once delivery was described as unachievable for many years. However, it is essential to many businesses where repeated data would be disastrous.

The best solution until AK 0.11.0 (CP 3.3) was to leverage an "at least once" delivery model and rely on end-user applications to do the deduplication after consumption.

# Overview of Exactly Once Semantics

- Fully supported on all versions of the Java clients and librdkafka-based clients (v.1.4.0 and later)
  - Producer and Consumer
  - Kafka Streams API
  - Confluent ksqlDB
  - Confluent REST Proxy
  - Kafka Connect
- **Transaction Coordinator:**
  - Broker thread that manages a special **transaction log**
- Transactions are made possible by a new message format:
  - **Sequence numbers** allow Brokers to skip duplicated messages
  - **Producer IDs** allow the Transaction Coordinator to prevent "zombie producers" from participating in a transaction

---

EOS was introduced as part of Kafka 0.11. It relies on a transaction coordinator to assign a set of ID numbers (Producer IDs, Sequence Numbers, Transaction IDs) and the clients to include this information in Magic Byte 2 message headers to uniquely identify messages.

EOS was released in conjunction with a major rewrite of the message and batching formats used by the Producers. As a result, users should see negligible differences in performance when coming from an older version to an EOS-enabled environment, despite the additional header fields and processing.

# Enabling Exactly Once Semantics

Two components:

## 1. Idempotent Producers

- Set `enable.idempotence = true` on the Producer

## 2. Transactions

- Set a unique `transactional.id` for each Producer
- Use the transactions API in the Producer code
- Set `isolation.level = read_committed` on the Consumer

---

EOS is enabled on the clients only—no changes need to be made to the Brokers other than ensuring that they are running Kafka 0.11.0 or later.

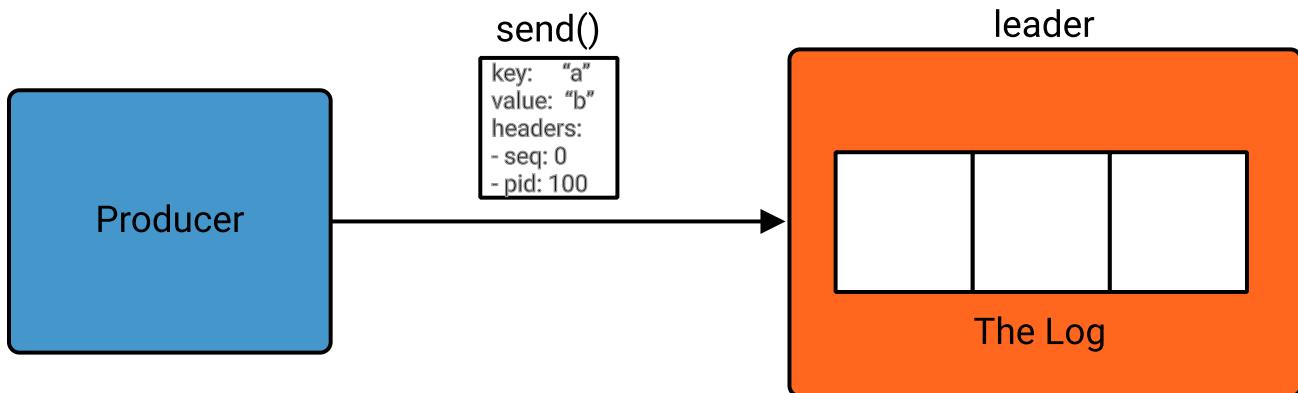
The `enable.idempotence = true` setting in the Producer ensures messages aren't duplicated, even in the case of Producer retries or Broker failure.



Enabling idempotent producers usually has negligible performance impact, thus making it useful in many non-transactional situations as well. The caveats to enabling idempotence are that `max.in.flight.requests.per.connection` must be less than or equal to 5, `retries` must be greater than 0 and `acks` must be "all". If these values are not explicitly set by the user, suitable values will be chosen. If incompatible values are set, a `ConfigException` will be thrown.

To do transactions, each Producer must have a unique `transactional.id` and use transaction-specific calls in the Producer API. The Consumer must also set `isolation.level = read_committed` so that it reads only committed transaction messages. This gives strong guarantees that transactional messages will be atomic.

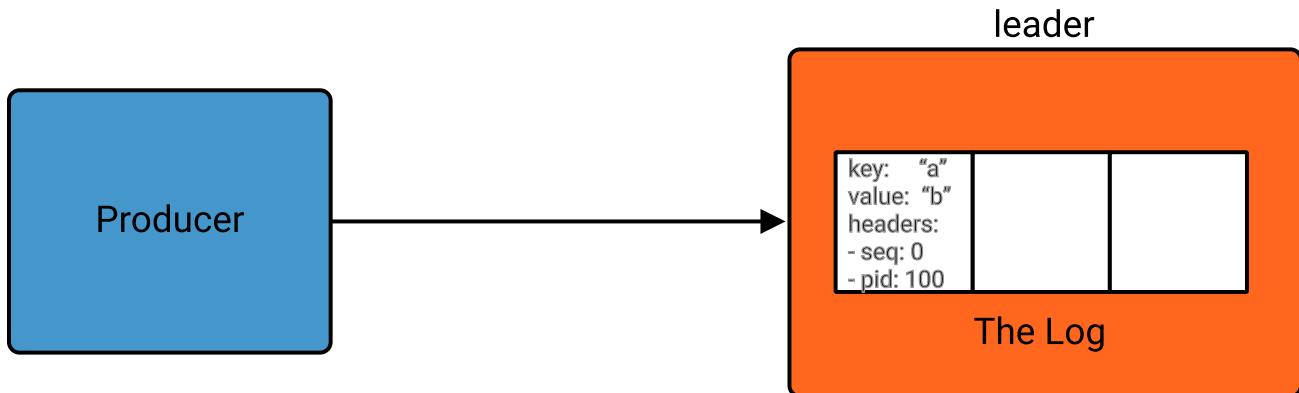
# Idempotent Producers (1)



Messages now have a sequence number and a Producer ID. A unique Producer ID is assigned for each Producer session.

tejaswin.renugunta@walgreens.com

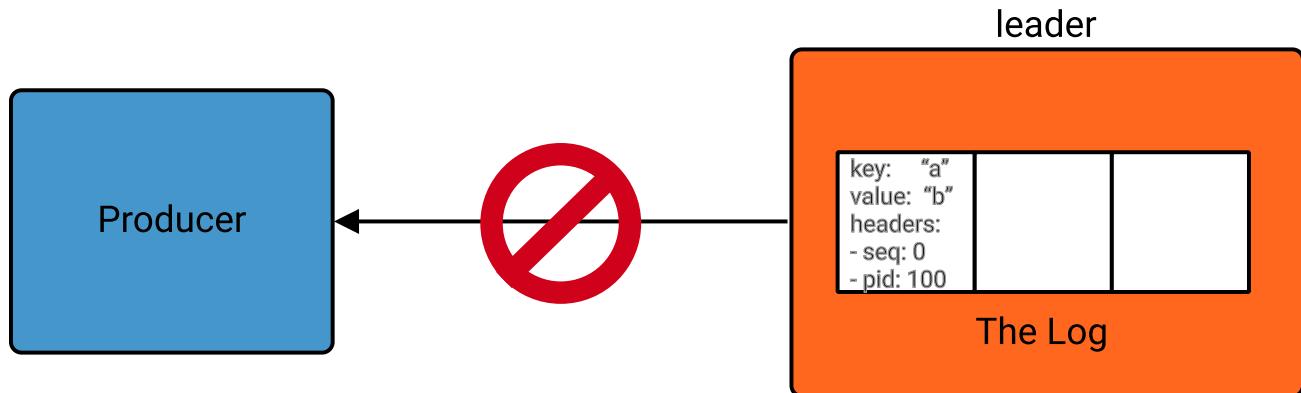
# Idempotent Producers (2)



The Broker will retain a map { PID : sequence number } in memory that is occasionally snapshotted to the log in a `*.snapshot` file. If the Broker recovers from failure, it could read through the log and catch up to the current mapping of PID → Sequence number, but this could take a while. The `*.snapshot` file speeds up this process.

tejaswin.renugunta@wipro.com

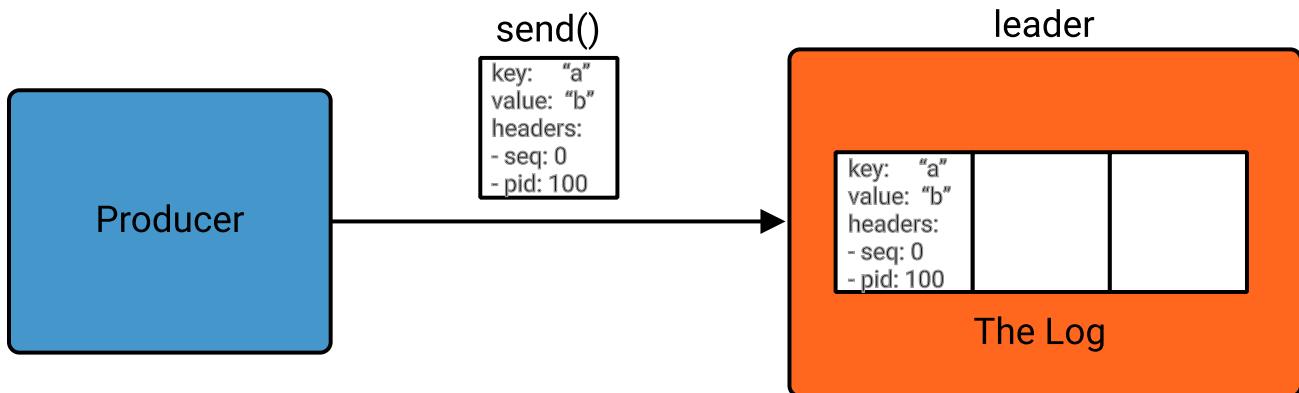
# Idempotent Producers (3)



In this scenario, the Broker fails to send an **ack** back to the Producer.

tejaswin.renugunta@walgreens.com

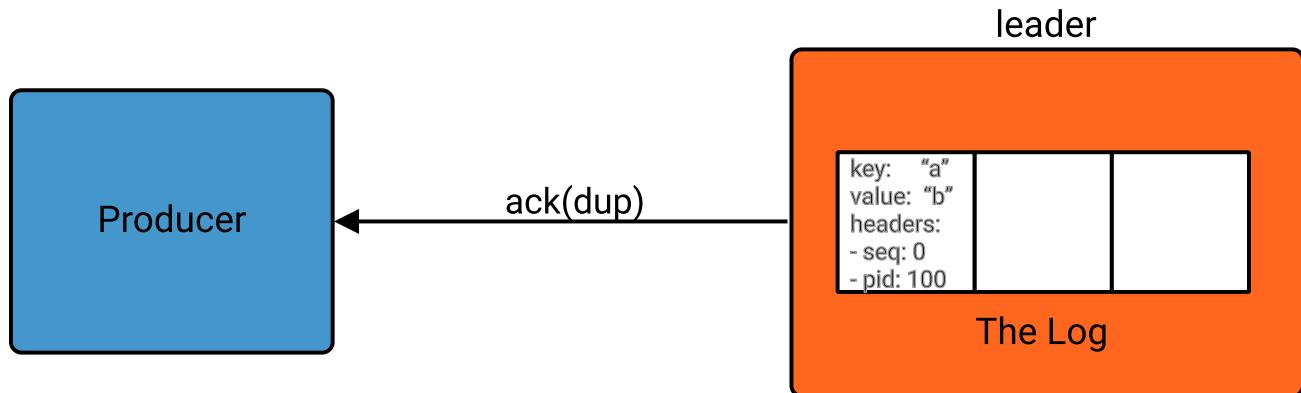
# Idempotent Producers (4)



The Producer **retries**. Without idempotence enabled, the message would be duplicated.

tejaswin.renugunta@walgreens.com

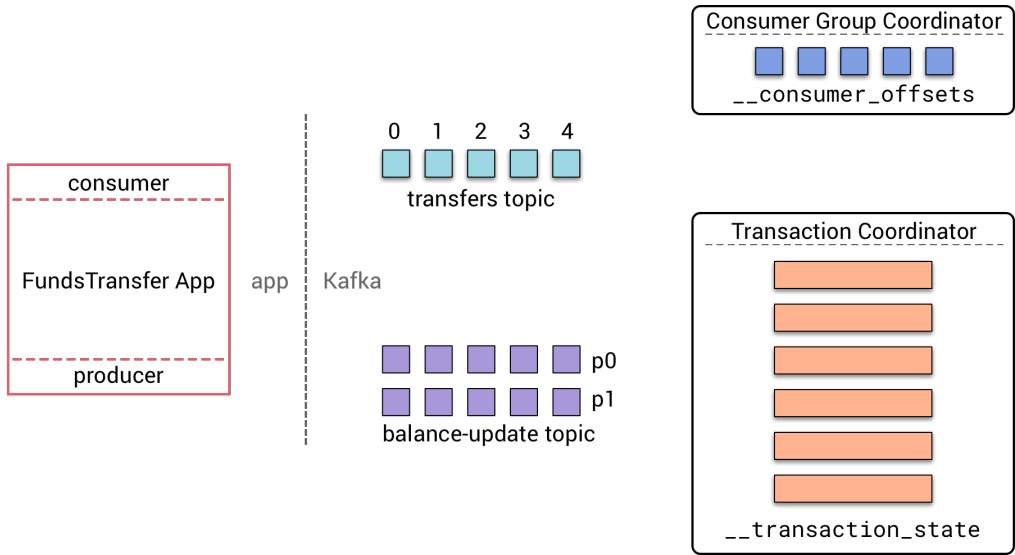
# Idempotent Producers (5)



Instead, the Broker checks the sequence number and when it sees the message is a duplicate, it will ignore the record and return a DUP response to the client.

tejaswin.renugunta@walgreens.com

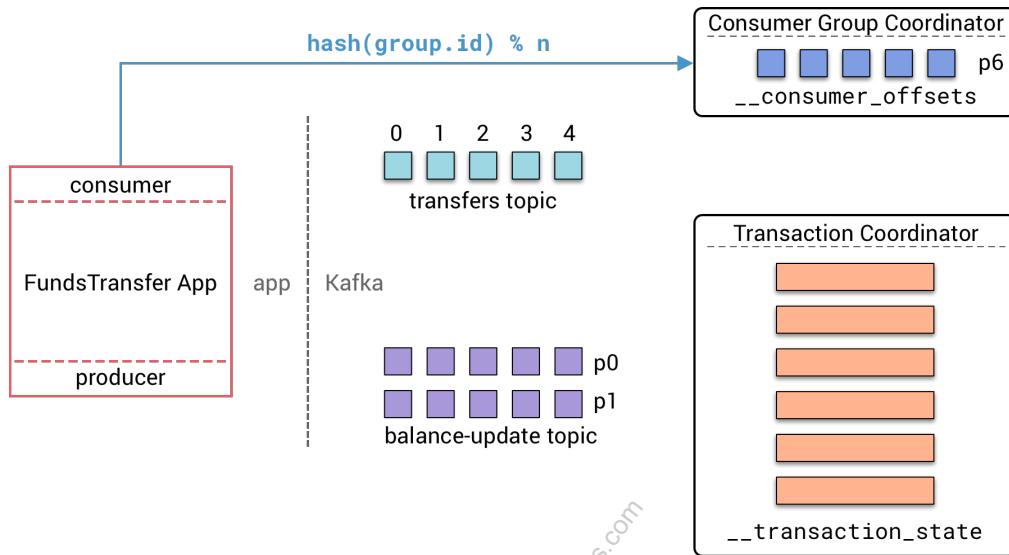
# Transactions (1/14)



Pictured is a stream processing application called FundsTransfer that follows the consume-process-produce paradigm. The idea is to read a financial transaction from the "transfers" topic and produce balance updates to the "balance-update" topic.

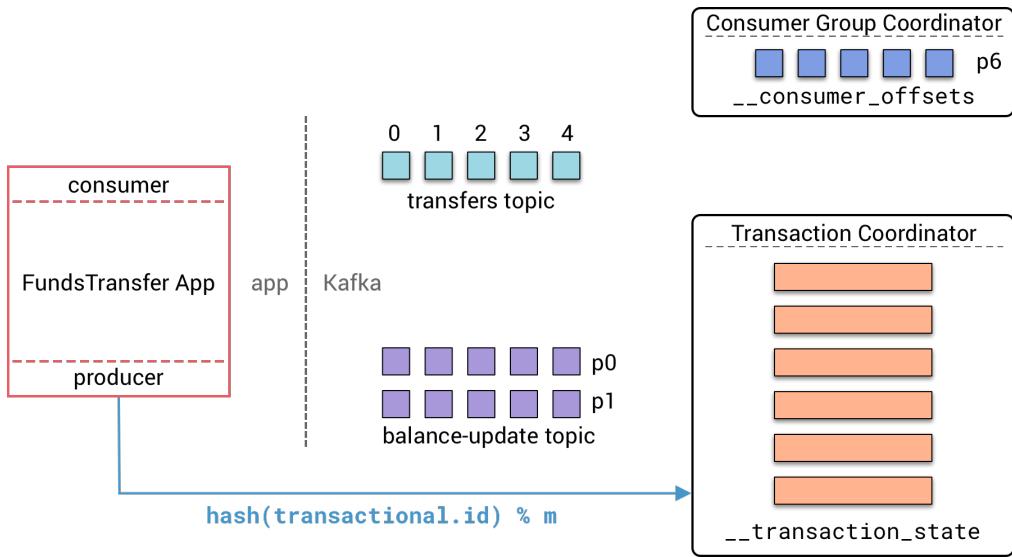
A Transaction Coordinator is a module that is available on any Broker. The Transaction Coordinator is responsible for managing the lifecycle of a transaction in the "Transaction Log"—the internal Kafka Topic `--transaction_state` partitioned by `transactional.id`. The Broker that acts as the Transaction Coordinator is not necessarily a Broker that the Producer is sending messages to. For a given Producer (identified by `transactional.id`), the Transaction Coordinator is the leader of the Partition of the Transaction Log where `transactional.id` resides. Because the Transaction Log is a Kafka Topic, it has durability guarantees.

# Transactions - Initialize Consumer Group (2/14)



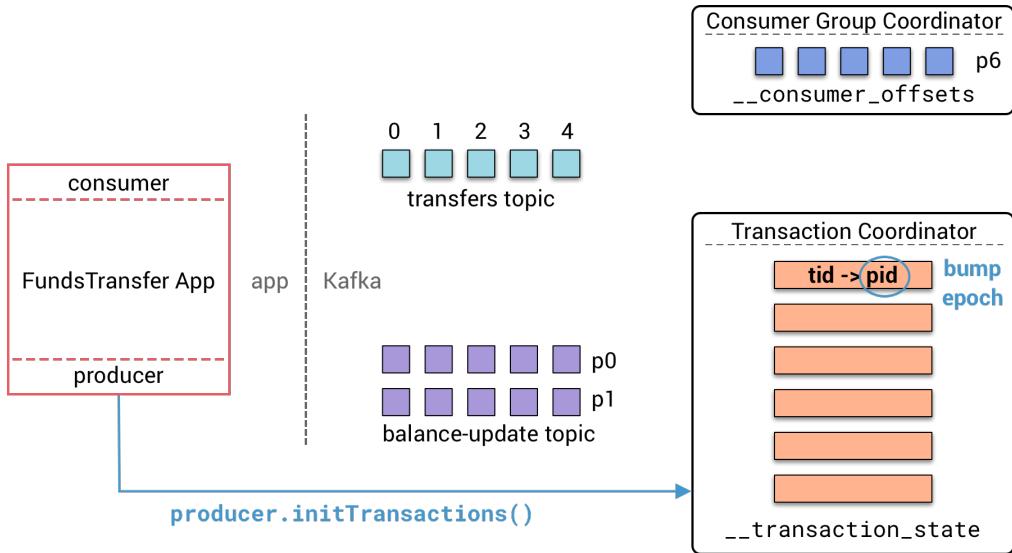
The consumer and producer are initialized before stream processing is started. Here we see the consumer subscribe to the "transfers" topic and identify its Consumer Group Coordinator using `hash(group.id) % n`, where `n` is the number of partitions of the consumer offsets topic (Default: 50). Here, the `p6` indicates that this Consumer Group Coordinator is the broker that holds the lead replica for partition 6 of the consumer offsets topic.

# Transactions - Transaction Coordinator (3/14)



Here we see the producer initiating the transaction. The producer identifies the Transaction Coordinator using `hash(transactional.id) % m`, where `m` is the number of partitions of the `--transaction_state` topic (Default: 50).

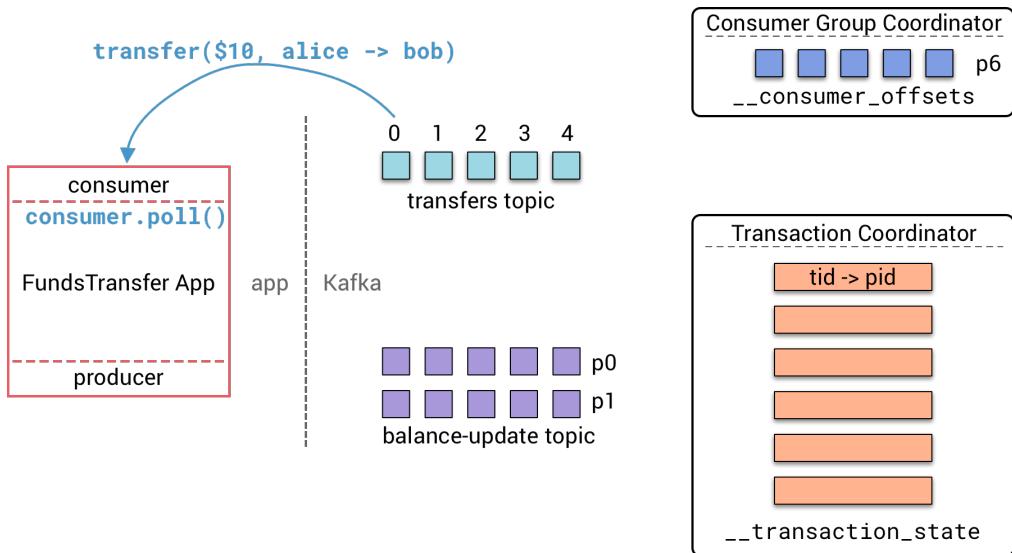
# Transactions - Initialize (4/14)



During the initiation, the Producer registers itself to the Transaction Coordinator with its `transactional.id`. The Transaction Coordinator records a mapping `{ Transactional ID : Producer ID }`. The Transaction Coordinator also increments an `epoch` associated with the `transactional.id`.

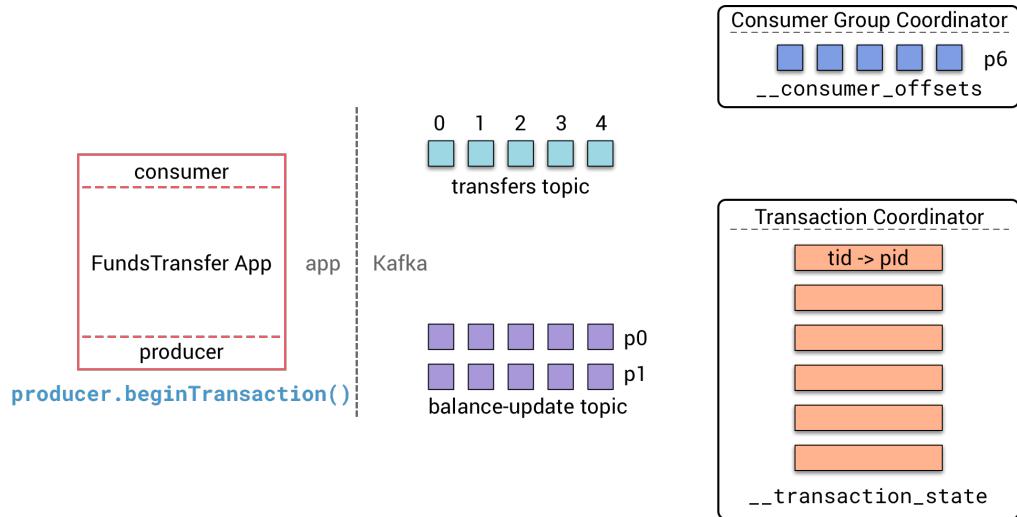
The epoch is an internal piece of metadata stored for every `transactional.id`. Once the epoch is bumped, any producers with same `transactional.id` and an older epoch are considered zombies and are fenced off and future transactional writes from those producers are rejected. This enables reliability semantics which span multiple producer sessions since it allows the client to guarantee that transactions using the same TransactionalId have been completed prior to starting any new transactions.

# Transactions - Consume and Process (5/14)



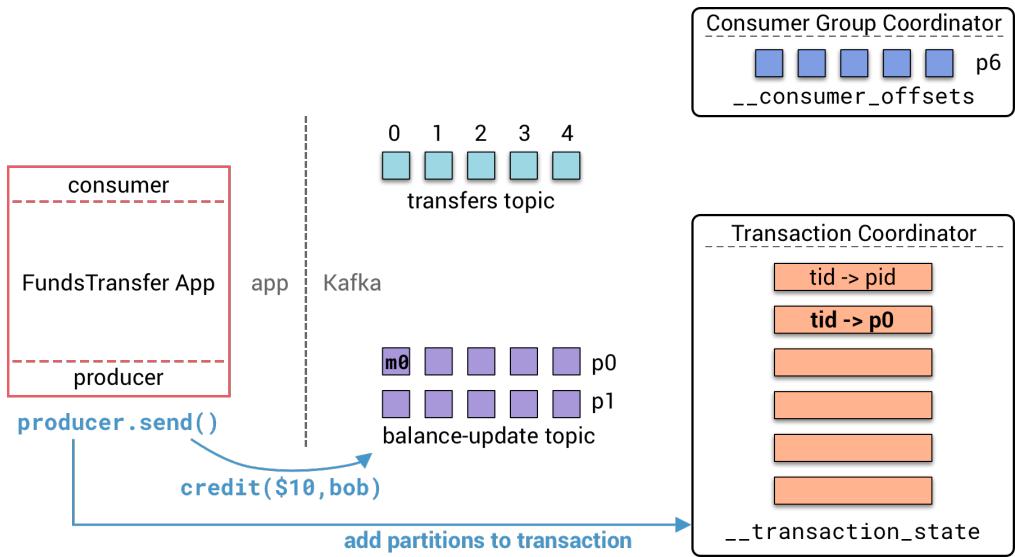
The Consumer polls for messages from the input Topic. Here, the consumer reads an event that transfers \$10 from Alice to Bob. The goal of the FundsTransfer app is to transactionally write events to the "balance-update" topic that credits Bob with \$10 and debits \$10 from Alice.

# Transactions - Begin Transaction (6/14)



The Producer begins the transaction.

# Transactions - Send (7/14)

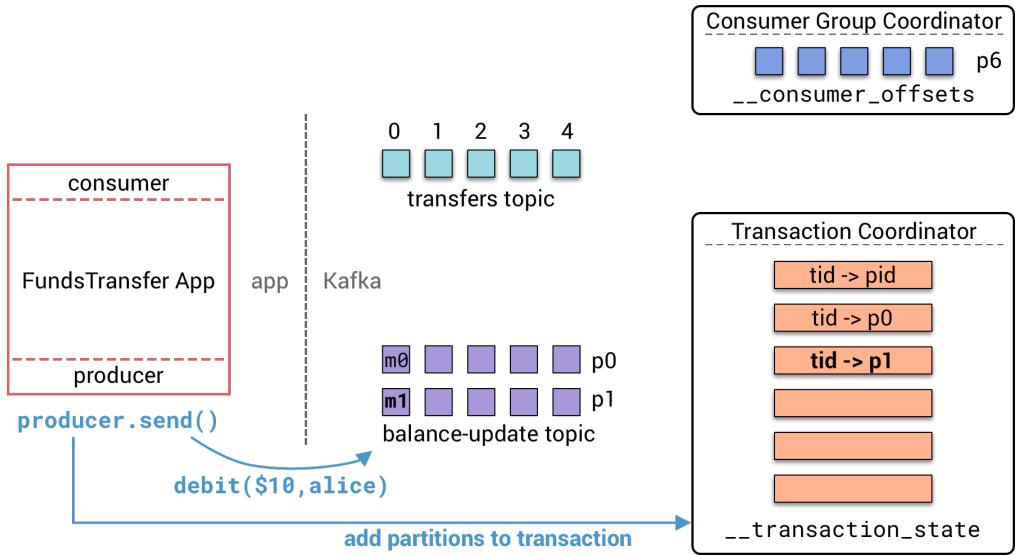


The Producer sends a message to a partition. Here, the message is to credit Bob with \$10.



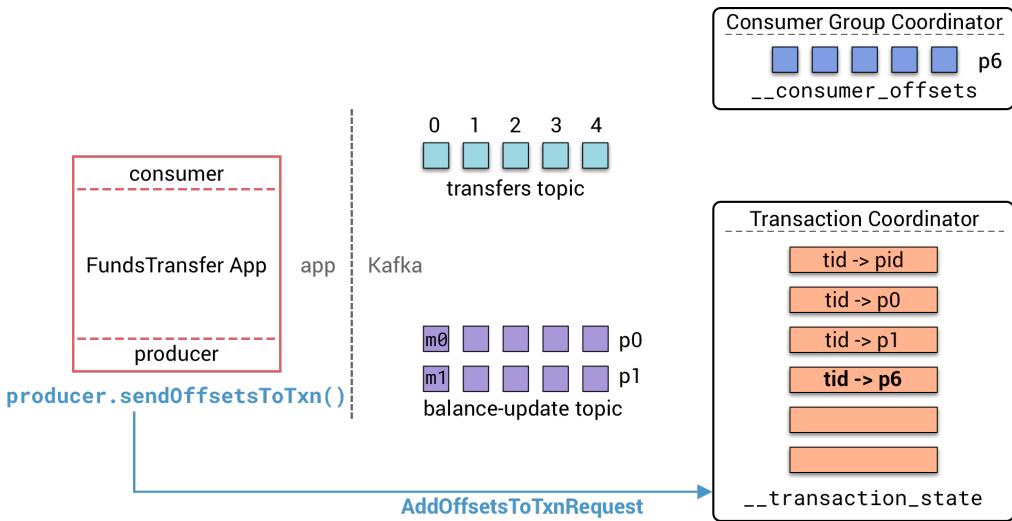
The first time a new TopicPartition is written to as part of a transaction, the producer sends a "Register Partitions" request to the transaction coordinator and this TopicPartition is logged. The transaction coordinator needs this information so that it can write the commit or abort markers to each TopicPartition. If this is the first partition added to the transaction, the coordinator will also start the transaction timer.

# Transactions - Send (8/14)



The Producer sends a message to a second partition. Here, the message is to debit \$10 from Alice. This message happens to land on a different partition from the previous message, so this partition is also added to the transaction log.

# Transactions - Track Consumer Offset (9/14)



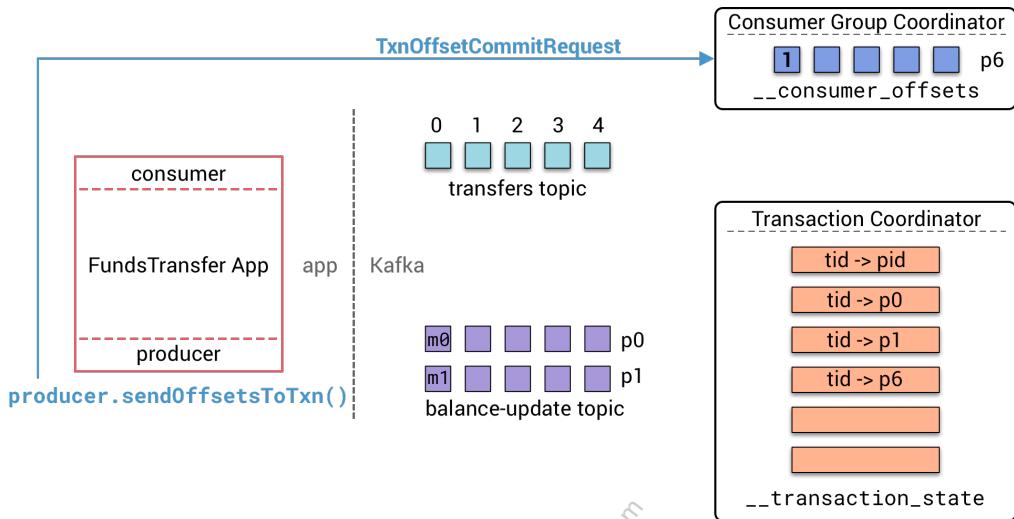
The `sendOffsetsToTxn()` method sends the consumer's offset and consumer group information to the transaction coordinator via an `AddOffsetCommitsToTxnRequest`. This makes the consumer's offset become a part of the transaction. If the transaction fails, the consumer's offset doesn't move forward and the transaction can start over.

Of course, the consumer may be subscribed to multiple partitions across multiple topics, in which case all relevant consumer offsets are included in the transaction state log. In this simple example, there is only one partition's offset to track.



To take advantage of the `sendOffsetsToTxn()` method, the consumer should have `enable.auto.commit=false` and should also not commit offsets manually. See [the Java API documentation](#)

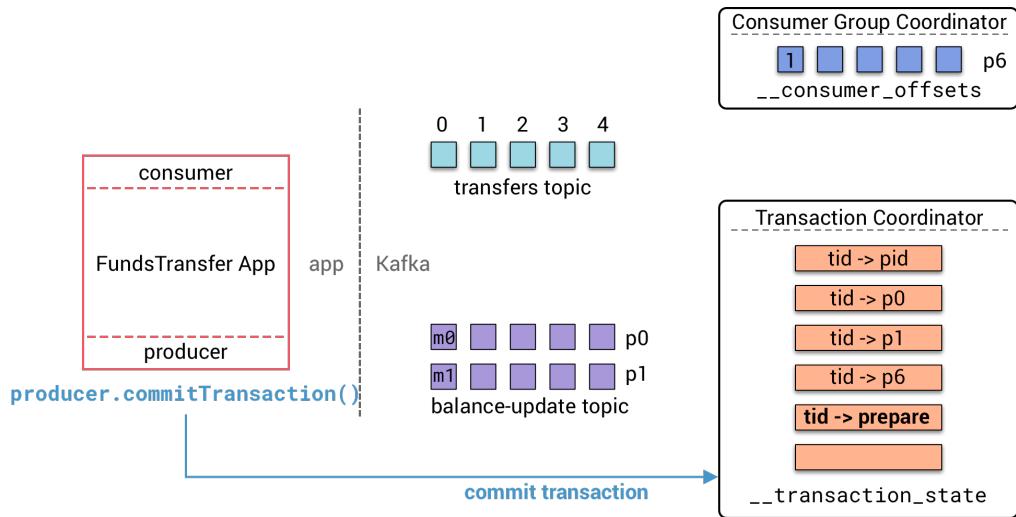
# Transactions - Commit Consumer Offset (10/14)



Also as part of `sendOffsetsToTxn()`, the producer will send a `TxnOffsetCommitRequest` to the consumer coordinator to persist the offsets in the `__consumer_offsets` topic.

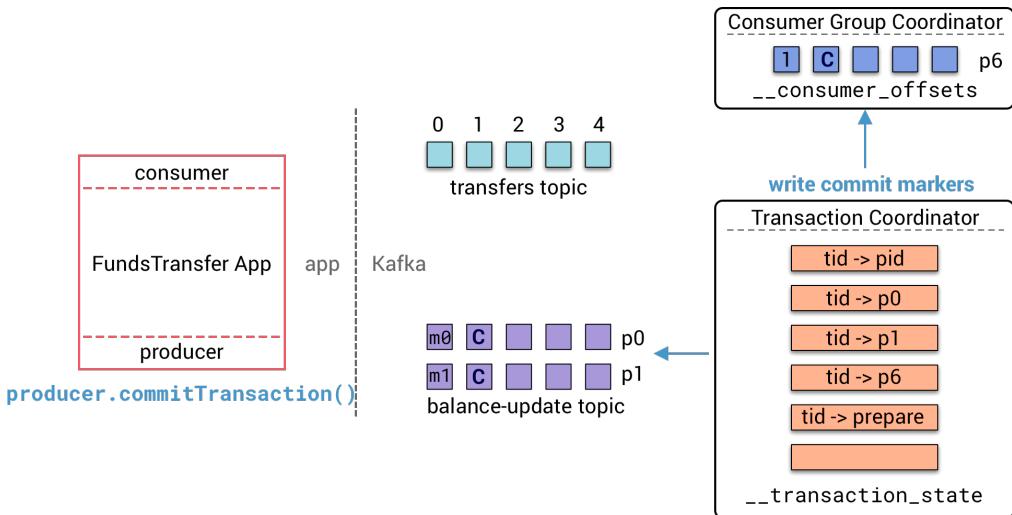
This guarantees the offsets and the output records will be committed as an atomic unit.

# Transactions - Prepare Commit (11/14)



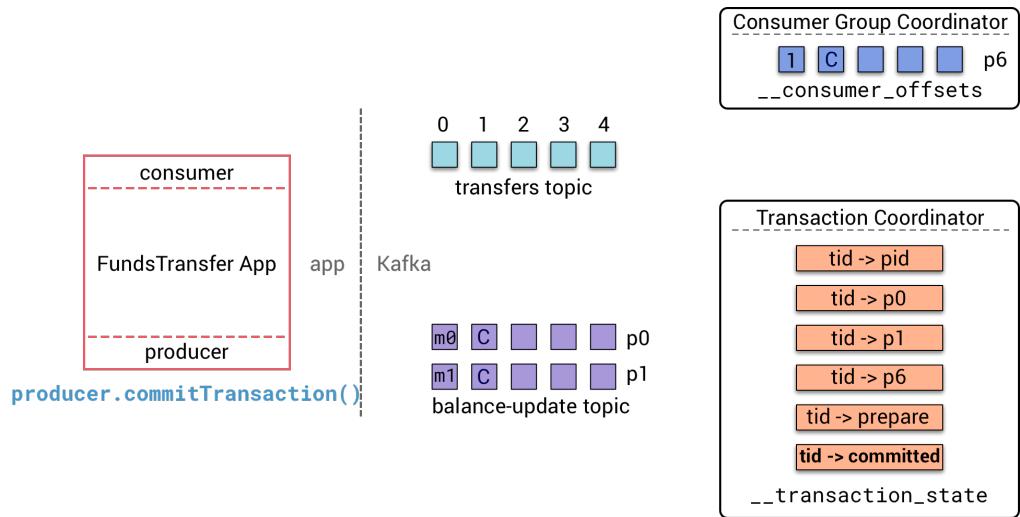
Producer commits the transaction. The transaction coordinator marks the transaction as in status of "preparing".

# Transactions - Write Commit Markers (12/14)



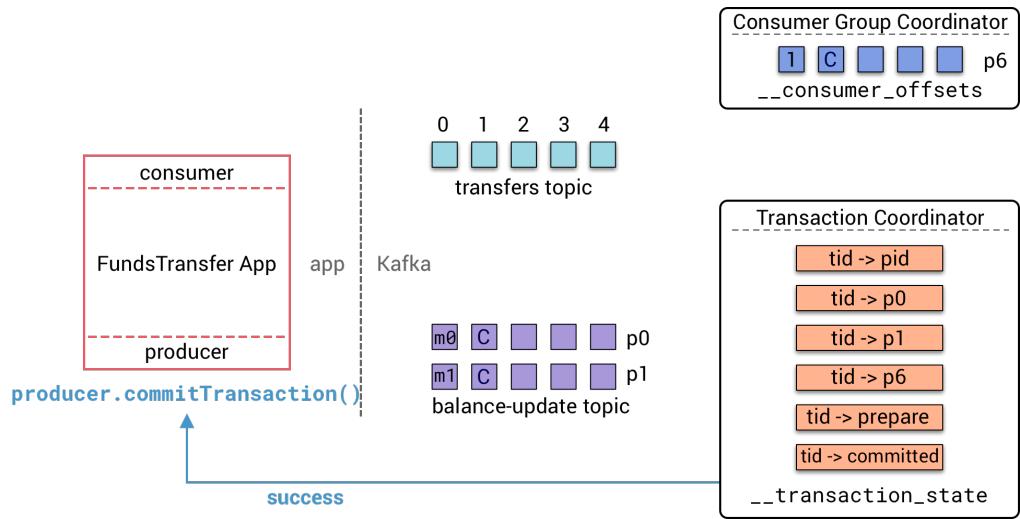
The Transaction Coordinator writes commit markers to the Partitions the Producer writes to as well as to the `__consumer_offsets` Partition. Commit markers are special messages which log the producer id and the result of the transaction (committed or aborted). These messages are internal only and are not exposed by standard consumer operations.

# Transactions - Commit (13/14)



The Transaction Coordinator marks the transaction as committed.

# Transactions - Success (14/14)



As a final step the transaction coordinator sends an acknowledgement to the producer.

# Consume Committed Transactions

- `isolation.level=read_committed`: reads only committed transactional messages and all non-transactional messages
- Consumer API alone cannot guarantee exactly-once **processing**
- Guarantee exactly-once processing with "consume-process-produce" pattern:
  - Set `enable.auto.commit=false` in Consumer
  - Use `sendOffsetsToTransaction()` in Producer
- The **Kafka Streams API** was designed with exactly-once processing in mind:
  - Set property `processing.guarantee=exactly_once`

---

Default behavior is for the Consumer to be set to `read_uncommitted`, which will read all messages regardless of their transaction result.

Each Partition maintains an "abort index" file with suffix `.txnidex` that gets cached on `read_committed` Consumers so they can quickly skip messages from aborted transactions.

	If a Producer dies in the middle of a transaction and a new Producer doesn't take its place, all <code>read_committed</code> Consumers must wait for the amount of time specified by the Producer property <code>transaction.timeout.ms</code> (default 60 sec) for the transaction to be aborted and the Last Stable Offset to advance before they can move forward in the log. Any transactional and non-transactional messages written to the log after the uncommitted transactional message(s) will not be consumed until this abort occurs.
---	---

EOS semantics only guarantee exactly once delivery into Kafka. On the Consumer side, To ensure transactional semantics for the "consume-process-produce" pattern, a client application should set `enable.auto.commit=false` and should not commit offsets manually, and instead use the `sendOffsetsToTransaction()` method in the `KafkaProducer` interface.

	EOS was designed primarily for the Kafka Streams API where consume-process-produce is the standard execution model, so Kafka Streams is highly recommended if creating applications that require exactly once processing.
---	---

# Hands-On Lab

- In this Hands-On Exercise, you will observe the Kafka distributed log; you will see how data is split by Partition and replica, and you will look at the contents of the log files. You will then see how Kafka dynamically recovers from machine failure
- Please refer to **Lab 03 Providing Durability**
  - a. **Investigating the Distributed Log**



tejaswin.renugunta@walgreens.com

# Module Review



- Kafka has a very durable architecture
- Data is committed and replicated
- The ISR list contains all In-Sync Replicas
- The cluster can manage and recover from failed Brokers
- It is possible to achieve transactions by using Exactly Once Semantics

tejaswin.renugunta@walgreens.com

# 04: Managing a Kafka Cluster



CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Providing Durability
4. Managing a Kafka Cluster ... ←
5. Optimizing Kafka's Performance
6. Kafka Security
7. Data Pipelines with Kafka Connect
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

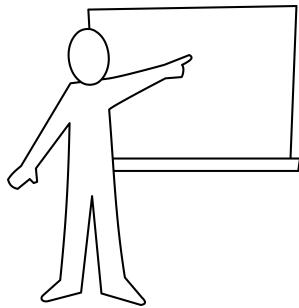


After this module you will be able to:

- Describe the basics of how to configure and run Kafka
- List the monitoring capabilities in Kafka
- Perform common cluster administration tasks
- Explain what log compaction is, and why it is useful
- Expand and shrink the cluster size

tejaswin.renugunta@walgreens.com

# Module Map



- Installing and Running Kafka ... ↵
- ⚙️ Hands-on Lab: **Exploring Configuration**
- ⚙️ Hands-on Lab: **Automating Configuration**
- Monitoring Kafka
- Managing Topic Configurations
- ⚙️ Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction
- An Elastic Cluster
- ⚙️ Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Manual Installation

- Confluent Platform includes all components
- Install on Linux via package manager
  - Components managed with `systemd` services or start/stop scripts
- ZooKeeper → Kafka → Schema Registry
- Separation of Concerns
- Create "chroot" path for Kafka cluster in ZooKeeper
- Choose a flavor of Java



On various Linux distributions (Ubuntu, Debian, RHEL, and CentOS), there is a typical process of installing via a package manager. The installation creates a user `cp-<component>` for each component, a group called `confluent`, and sets permissions for these users on the configured data directories. This encourages the best practice of "Principle of Least Privilege." Here is an example of manual installation and configuration in Debian based Linux distributions:

[https://docs.confluent.io/current/installation/installing\\_cp/deb-ubuntu.html](https://docs.confluent.io/current/installation/installing_cp/deb-ubuntu.html).

Manual installation via `zip` or `tar` archives is also available:

[https://docs.confluent.io/current/installation/installing\\_cp/zip-tar.html#prod-kafka-cli-install](https://docs.confluent.io/current/installation/installing_cp/zip-tar.html#prod-kafka-cli-install).

For development purposes, you can install Confluent Platform and use the `confluent` CLI (<https://docs.confluent.io/current/cli/>) or run Docker-Compose locally (<https://docs.confluent.io/current/quickstart/ce-docker-quickstart.html#cp-quick-start-docker>).

Confluent Platform includes `start/stop` scripts for services, and also includes `systemd` unit files to manage components with `systemctl`. ZooKeeper, Kafka, and Schema Registry must be started in that order because Kafka stores cluster metadata in ZooKeeper and Schema Registry stores its state in Kafka.

To adhere to the "Separation of Concerns" principle, multiple components should not be run on the same machine. Just download the entire package and activate the component

appropriate for that system.

CP supports Java 8 and 11 (Java 11 added in CP 5.2). As of Apache Kafka 1.0, Java 9 is supported but not by Confluent Community components like Schema Registry or REST Proxy. For more complete information about supported versions and interoperability, see: <https://docs.confluent.io/current/installation/versions-interoperability.html>.

There is some light configuration of ZooKeeper required via the `zookeeper.properties` file (see e.g. [https://docs.confluent.io/current/installation/installing\\_cp/rhel-centos.html#zk](https://docs.confluent.io/current/installation/installing_cp/rhel-centos.html#zk)). In some environments, ZooKeeper is used for more than just Kafka (or used for multiple Kafka clusters). If this is the case, it is customary to configure a special "chroot" path for your Kafka cluster with

```
$ zookeeper-shell create /<cluster-name> []
```

If this is the case, then the `zookeeper.connect` property in each Broker's `server.properties` file must be modified to store cluster metadata at the `<cluster-name>` namespace. For example,

```
zookeeper.connect=zk-1:2181/<cluster-name>,zk-2:2181/<cluster-name>,zk-3:2181/<cluster-name>
```

To be clear, this must be done after configuring and starting the ZooKeeper service but before starting the Kafka service.

# Running Services

- ZooKeeper:

```
$ /usr/bin/zookeeper-server-start /etc/kafka/zookeeper.properties  
$ /usr/bin/zookeeper-server-stop
```

- Kafka Broker:

```
$ /usr/bin/kafka-server-start /etc/kafka/server.properties  
$ /usr/bin/kafka-server-stop
```



Because the exercises in this class are run on Docker containers, the lab environment for this course does not use a standard `server.properties` file. The Broker configurations are set via the `docker-compose.yml` file. By way of a script, properties defined in the `docker-compose.yml` file are placed in `/etc/kafka/kafka.properties` inside of each Broker container.

# Running Confluent Platform Services

- Confluent Platform `systemd` services with `systemctl`:

```
$ sudo systemctl start confluent-<component>
$ sudo systemctl stop confluent-<component>
$ sudo systemctl enable confluent-<component> # enable service
on startup
```

---

CP began including `systemd` unit files in CP 4.1.

tejaswin.renugunta@walgreens.com

# Configuring the Cluster Properties

- **Brokers:** modify `/etc/kafka/server.properties` before (re)starting the Broker
- **Topics:** use `--config` option

```
$ kafka-topics --bootstrap-server broker-1:9092 \
  --create
  --topic i-love-kafka \
  --partitions 5
  --replication-factor 3 \
  --config min.insync.replicas=2 cleanup.policy=compact
```

- **Producers, Consumers:** client code
- Dynamic changes on **Brokers** or **Topics** with `kafka-configs` command

---

The `server.properties` file is used to set Broker-specific settings (`broker.id`) as well as cluster defaults (`default.replication.factor`). If the cluster defaults are not consistent across the Brokers in the cluster, you may experience unpredictable behavior.

Confluent Control Center (a.k.a. CCC or C3) provides a dashboard to inspect Broker configurations.

Any change to `server.properties` requires a reboot of the Broker to take effect. However, some of the Broker properties can be changed dynamically. The Configurations section of the documentation page listed on the slide includes a column which indicates which Broker properties are dynamically configurable.

Here is an example of a `server.properties` file for a production Kafka cluster for reference:

```
# ZooKeeper
zookeeper.connect=[list of ZooKeeper servers]

# Log configuration
num.partitions=8
default.replication.factor=3
log.dirs=[List of directories. Kafka should have its own dedicated disk or SSD per directory.]

# CPU thread configurations
num.io.threads =[one per log directory]
num.network.threads =[increase from 3 for TLS]
num.replica.fetchers =[increase from 1 for many replicas]
log.cleaner.threads =[increase if log cleaning is IO bound]

# Other configurations
broker.id=[An integer. Start with 0 and increment by 1 for each new Broker.]
listeners=[list of listeners]
advertised.listeners=[list of listeners to publish to ZooKeeper for external clients]
auto.create.topics.enable=false
min.insync.replicas=2
queued.max.requests=[number of concurrent requests expected]
```

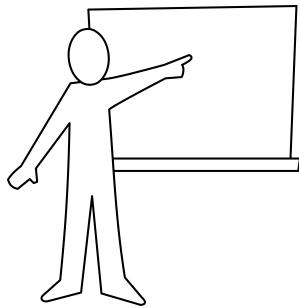
For more configuration information, see

<https://docs.confluent.io/current/installation/configuration/index.html>.



The `--bootstrap-server` option for `kafka-topics` was introduced in Apache Kafka 2.2. Previously, the `--zookeeper` option was required.

# Module Map



- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration ...** ←
- Hands-on Lab: **Automating Configuration**
- Monitoring Kafka
- Managing Topic Configurations
- Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction
- An Elastic Cluster
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Hands-On Lab

- In this Hands-On Exercise you will explore Kafka Broker configuration properties using the Confluent Control Center
- Please refer to **Lab 04 Managing a Kafka Cluster** in Exercise Book:
  - a. **Exploring Configuration**



tejaswin.renugunta@walgreens.com

# Automated Configuration

- Confluent Platform Ansible Playbook
- Confluent Operator
  - Official Kubernetes operator



---

A best practice is to use a configuration management tool (e.g. Ansible, Helm, Puppet, Chef) to automate configuration. This approach allows administrators to apply sound software development practices like version control, continuous integration, and continuous deployment. Confluent provides several configuration management options with the CP Ansible Playbook, CP Kubernetes Helm Chart, and Confluent Kubernetes Operator.

The CP Ansible Playbook is ideal for a more traditional environment with physical or virtual machines. It only requires `ssh` access to instances. The Playbook can be deployed with only a few changes to conform to the user's environment.

There are some finer points of Kafka operations in Kubernetes that are addressed with the Confluent Operator. Kafka is inherently a stateful technology, so some special care must be taken. Failure recovery requires recovering the identity and data of the failed Broker. Scaling requires intelligent Partition rebalancing. Rack awareness needs to play into how Kubernetes schedules Broker pods. Rolling restarts need to be sensitive to under-replicated Partitions. Some software updates require restarting twice. All of these Kafka-specific management considerations and more are done automatically with the Confluent Operator in response to changes in declarative `yml` files.

Here are some resources for further information:

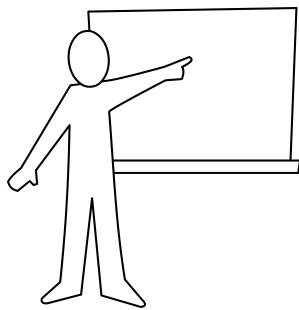
- CP Ansible Playbook: <https://docs.confluent.io/current/tutorials/cp-ansible/docs/index.html#cp-ansible>
- Kubernetes Helm Charts: [https://docs.confluent.io/current/installation/installing\\_cp/cp-helm-charts/docs/index.html#kubernetes-helm-charts](https://docs.confluent.io/current/installation/installing_cp/cp-helm-charts/docs/index.html#kubernetes-helm-charts)
- Confluent Kubernetes Operator:  
<https://docs.confluent.io/current/installation/operator/index.html>



The default value of `broker.id.generation.enable` is `true`, so it's possible to manually add Brokers to the cluster without setting `broker.id`. ZooKeeper will automatically generate unique Broker IDs starting from `reserved.broker.max.id + 1` (Default 1001, 1002, etc.). This is useful in some situations, but shouldn't be necessary when using a configuration management tool because the tool can iterate Broker IDs.

tejaswin.renugunta@walgreens.com

# Module Map



- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration**
- Hands-on Lab: **Automating Configuration ...** ↩
- Monitoring Kafka
- Managing Topic Configurations
- Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction
- An Elastic Cluster
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Hands-On Lab

- In this Hands-On Exercise you will explore Kafka Broker configuration properties using the Confluent Control Center
- Please refer to **Lab 04 Managing a Kafka Cluster** in the Exercise Book:
  - a. **Automating Kafka Configuration**



tejaswin.renugunta@walgreens.com

# Updating Broker Configurations

- Broker configuration types:
  - **per-broker**: may be updated dynamically for each Broker
  - **cluster-wide**: may be updated dynamically for *all* Brokers
  - **read-only**: requires a broker restart for update
- Order of precedence:
  1. Dynamic per-broker config in ZooKeeper
  2. Dynamic cluster-wide default config in ZooKeeper
  3. Static broker config in **server.properties**
  4. Kafka default

---

Dynamic changes will take effect immediately, but will not be updated in the **server.properties** file. However, the changes will persist because the **kafka-configs** command updates the configurations in ZooKeeper.

If using configuration management tools, the best practice is to change configurations in code so that they can be version controlled and then applied as a part of a CI/CD pipeline. Using **kafka-configs** to change configuration dynamically has the benefit of not requiring service restart (and thus avoiding some increased replication bandwidth and data loss for unreplicated Topics), but it forgoes the benefits of adhering to modern change management systems.

# Changing Broker Configurations Dynamically

## (1)

- Display dynamic Broker configurations for Broker with Broker ID **103**:

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker 103 \
  --describe
```

- Change a cluster-wide default configuration

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker-defaults \
  --alter \
  --add-config log.cleaner.threads=2
```

- To alter multiple config settings, use **--add-config-file new.properties**

---

The default **--describe** behavior for the **kafka-configs** command is to list dynamic config settings for the current entity. To display all config settings, specify **--all**.

**log.cleaner.threads** tunes how many CPU threads to dedicate to log cleaning. Log cleaning will be discussed later in the course.

# Changing Broker Configurations Dynamically (2)

- Change a Broker configuration

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker 101 \
  --alter \
  --add-config log.cleaner.threads=2
```

- Delete a Broker configuration

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker 101 \
  --alter \
  --delete-config log.cleaner.threads
```

"Delete a Broker configuration" just means that you are resetting it to the cluster default.

tejaswin.renugunta@valgreens.com

# Upgrading a Cluster



- Rolling Broker upgrades:
  - back up configuration files from `/etc`
  - stop service
  - uninstall the old Confluent Platform
  - install new Confluent Platform
  - start service
  - Some updates require special steps
- Clients and Brokers can be upgraded independently

---

Upgrading to a new version of Kafka can require special steps if there are fundamental changes to Kafka like the message format or inter Broker protocol. To upgrade in these cases (for example, upgrading to Kafka 2.2 from any older version), one must set `inter.broker.protocol.version` and `log.message.format.version` equal to the current Kafka version in the `server.properties` file.

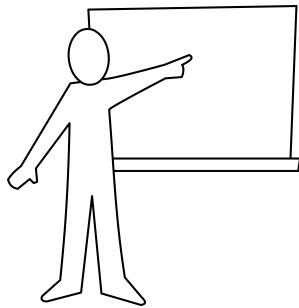
Refer to the documentation for more specific instructions:

<https://docs.confluent.io/current/installation/upgrade.html>.

If using a configuration management tool, rolling restarts must be taken into account. As of CP 5.2, the CP Ansible Playbook does not support graceful rolling restarts. At minimum, the playbook would need to be adjusted to put in adequate sleep time between configuring hosts to allow for followers to catch up to leaders.

As aforementioned, the Confluent Kubernetes Operator automatically performs a graceful rolling restart in response to applying a change.

# Module Map



- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration**
- Hands-on Lab: **Automating Configuration**
- Monitoring Kafka ...
- Managing Topic Configurations
- Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction
- An Elastic Cluster
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Monitoring Your Kafka Deployments

- You can use Confluent Control Center to
  - Optimize performance
  - Verify Broker and Topic configurations
  - Identify potential problems before they happen
  - Troubleshoot issues
- What else to monitor
  - Kafka logs
  - Kafka metrics (**JMX**)
  - System logs
  - System resource utilization

---

JMX (Java Management Extensions) is a Java technology used for monitoring applications. The **JmxReporter** Java class is always included to register JMX statistics. Other custom reporter classes can be plugged in by adding the **.jar** file to Kafka's **CLASSPATH** (e.g. `/usr/share/java/kafka/`) and setting the **metric.reporters** Broker property to use the custom class. This requires a restart since a new class is added to the JVM.

If you ever run into problems with a Kafka deployment, whether you reach out to the community or to Confluent for support, you may be asked for relevant metrics. It is better to already be running with monitoring enabled.

# Logs vs. Logs

- Kafka Topic data
  - `log.dirs` property in `server.properties`
- Application logging with Apache `log4j`
  - `LOG_DIR`: configure the `log4j` files directory by exporting the environment variable (Default: `/var/log/kafka`)

---

We often refer to Topic data in Kafka as a log because it is philosophically an append-only log of events, but Kafka also requires application level logging (e.g. `TRACE`, `DEBUG`, `INFO`, `ERROR`, `FATAL`). Kafka uses the Apache `log4j` for application-level logging.

The location for Kafka Topic data is set using the `log.dirs` property which has a default value of `null`. If `log.dirs` is not set, then the location is set by the `log.dir` property which has a default value of `/tmp/kafka-logs`. In the lab environment for this class, `log.dirs` is set to `/var/lib/kafka`.

# Important log4j Files

- By default, the Broker `log4j` log files are written to `/var/log/kafka`
    - `server.log`: Broker configuration properties and transactions
    - `controller.log`: all Broker failures and actions taken because of them
    - `state-change.log`: every decision Broker has received from the Controller
    - `log-cleaner.log` : compaction activities
    - `kafka-authorizer.log` : requests being authorized
    - `kafka-request.log` : fetch requests from clients
  - Manage logging via `/etc/kafka/log4j.properties`
- 

Configurations possible via the `log4j.properties` file:

- Format how the date appears in the logs
- Set destination path for log files
- Change logging level from `WARN` to `TRACE` to troubleshoot

The `kafka-request.log` file tracks every request and includes the metadata of the request but not the messages themselves.

Due to the volume of data generated by `kafka-authorizer.log` and `kafka-request.log`, both are set to `WARN` by default.

# Tools for Collecting Metrics

- **Confluent Control Center**
- Other metrics tools:
  - JConsole
  - Graphite
  - Grafana
  - CloudWatch
  - DataDog



---

Kafka has metrics that can be exposed and inspected through clients. This is accomplished with a combination of Yammer and internal Kafka metrics packages.



Confluent does not specifically endorse any of the listed clients other than Confluent Control Center.

# Configuring the Cluster for Monitoring

- Enable JMX metrics by setting `JMX_PORT` environment variable

```
$ export JMX_PORT=9990
```

- Configure `client.id` on Producers and Consumers
  - Monitor by application
  - Used in logs and JMX metrics

---

The `client.id` is not required but can be used to represent one or more clients. It is usually set to identify separate applications to allow for more granular monitoring. It is used for monitoring (in JMX metrics and logs) and for bandwidth control (in the quotas feature of Kafka).

tejaswin.renugunta@walgreens.com

# Monitoring Kafka at the OS Level

```
1 [|||||          13.2%] 5 [|||      5.9%
2 [           0.0%] 6 [           0.0%
3 [|||         9.8%] 7 [|||      3.9%
4 [           0.0%] 8 [           0.7%
Mem[||||||||||||| 15.84G/16.0G] Tasks: 365, 947 thr; 3 running
Swp[|||||          102M/1.00G] Load average: 2.18 1.86 1.76
Uptime: 03:02:50

PID USER    PRI  NI   VIRT   RES S CPU% MEM% TIME+ Command
454 chuck.lar 17  0 5106M 127M W  4.9  0.8 2:15.13 /Applications/iTerm.app/C
22276 chuck.lar 17  0 4286M 10992 ?  3.1  0.1 0:00.37 /usr/sbin/screencapture -
660 chuck.lar 24  1 4339M 34356 ?  1.3  0.2 1:01.87 /Library/Bitdefender/AVP/
4347 chuck.lar 17  0 4862M 123M ?  1.2  0.8 0:07.19 /Applications/Google Chro
22277 chuck.lar 17  0 4820M 33756 ?  0.7  0.2 0:00.31 /System/Library/CoreServi
3842 chuck.lar 17  0 5181M 147M ?  0.6  0.9 1:51.64 /Applications/Visual Stud
429 chuck.lar  8  0 4907M 197M ?  0.3  1.2 4:14.34 /Applications/Google Chro
3786 chuck.lar 24  0 6009M 202M ?  0.3  1.2 1:52.79 /Applications/Visual Stud
22253 chuck.lar 24  0 4223M 4444 R  0.2  0.0 0:00.18 htop
3386 chuck.lar 17  0 4865M 209M ?  0.1  1.3 0:44.16 /Applications/Google Chro
506 chuck.lar  8  0 4526M 49116 ?  0.1  0.3 0:35.50 /Applications/Google Chro
543 chuck.lar 17  0 5464M 74856 ?  0.1  0.4 0:03.22 /System/Library/CoreServi
743 chuck.lar 24  0 5905M 270M ?  0.0  1.6 3:09.05 /Applications/Slack.app/C
651 chuck.lar 17  0 4898M 142M ?  0.0  0.9 0:20.56 /Applications/Google Chro
434 chuck.lar 24  0 5771M 90156 ?  0.0  0.5 2:24.21 /Applications/Visual Stud
593 chuck.lar 17  0 4790M 95644 ?  0.0  0.6 0:12.61 /Applications/Google Chro
565 chuck.lar 17  0 4786M 14144 ?  0.0  0.1 0:00.73 /System/Library/Input Met
3790 chuck.lar 16  0 5150M 140M ?  0.0  0.9 1:00.08 /Applications/Visual Stud
464 chuck.lar 24  0 5724M 99804 ?  0.0  0.6 2:05.65 /Applications/Slack.app/C
468 chuck.lar 17  0 4981M 30328 ?  0.0  0.2 0:04.59 /System/Library/CoreServi
3135 chuck.lar 17  0 4279M 26892 ?  0.0  0.2 0:02.18 /System/Library/CoreServi
505 chuck.lar  0  0 4808M 127M ?  0.0  0.8 1:52.61 /Applications/Google Chro
428 chuck.lar 24  0 4298M 30116 ?  0.0  0.2 0:04.72 /usr/libexec/trustd --age
F1:help F2:setup F3:search F4:filter F5:tree F6:sortByF7:Nice -F8:Nice -F9:kill F10:quit
```

- Open file handles
  - Set `ulimit -n 1000000`
  - Alert at 60% of the limit
- Disk
  - Alert at 60% capacity
- Network bytesIn/bytesOut
  - Alert at 60% capacity

Beyond monitoring CPU and memory usage, it is especially important to monitor the number of open file handles, disk capacity, and network bandwidth on any machine running as a Kafka Broker.

- If you run out of file handles, it's a hard failure.
- Alerting at 60% utilization of disk space gives you time to provision new hardware
- Network congestion can lead to ISR drops and increased latency, which could lead to request timeouts
- Network congestion on all Brokers typically means it is time to add more Brokers

Why the conservative 60% threshold? This allows bandwidth for infrequent but high traffic operations (e.g., rebalancing, recovery from Broker failure) as well as to have failover capacity in the event that a Broker is offline. Additionally, alerting at 60% provides teams enough time to fix the issues before they reach 100% utilization and have worse problems.

Students may ask for Linux measurement tool recommendations. Some of the more common ones include:

- CPU, memory—`top`, `htop`
- Open file handles—`cat /proc/sys/fs/file-nr`, `lsof`
- Disk space—`df -H`
- Network—`iftop`, `iperf`

- Disk IO—`iostop`, `iostat`



When installing Confluent Platform version 5.3 or greater, the system's open file handle limit is set to 100,000. This limit can be set manually with `ulimit -n <large-number>`. Kafka requires a very large number of open files at once.

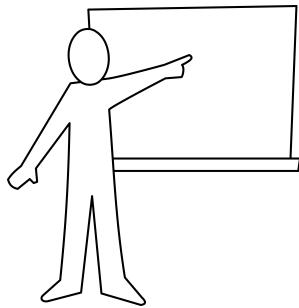
tejaswin.renugunta@walgreens.com

# Troubleshooting Issues

- Parse the `log4j` logs
  - Check metrics
  - Avoid unnecessary restarts
- 
- If needed, enable more detailed level of logging in `log4j.properties` (e.g. `WARN` → `TRACE`)
    - Changing this properties file requires a Broker restart unless using `jconsole`
  - Check metrics
    - General Kafka metrics
    - Specific Producers, Consumers, Consumer Groups, Streams
    - System resource utilization
  - Check end-to-end metrics in Confluent Control Center (part of Confluent Enterprise)
  - Do not troubleshoot problems by just rebooting nodes to see if the problem "goes away"
    - A lot happens when a Broker goes offline, e.g. Leader elections, replica movement
    - Extra load is put on the other Brokers (CPU, memory, disk utilization)
    - Leaders may not be In-Sync with preferred replicas

The reference to "end-to-end metrics in Confluent Control Center" requires interceptors to be installed in Producers and Consumers to track messages from creation to consumption. That subject is beyond the scope of the course, but you can read more about it here:  
<https://docs.confluent.io/current/control-center/docs/installation/clients.html>

# Module Map



- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration**
- Hands-on Lab: **Automating Configuration**
- Monitoring Kafka
- Managing Topic Configurations ...
- Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction
- An Elastic Cluster
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Topic Configuration Overrides

- Broker-level default topic configurations
    - static: `server.properties`
    - dynamic: `kafka-configs --describe`
  - Topic level configurations to override Broker defaults
    - Example: `segment.bytes` overrides `log.segment.bytes`
- 

Broker defaults for topic configuration properties are set statically using `server.properties` and dynamically using the `kafka-configs` command. To view current dynamic settings, use the following commands:

- For dynamic settings that apply to all brokers:

```
kafka-configs --bootstrap-server <broker>:<port> --broker-defaults  
--describe
```

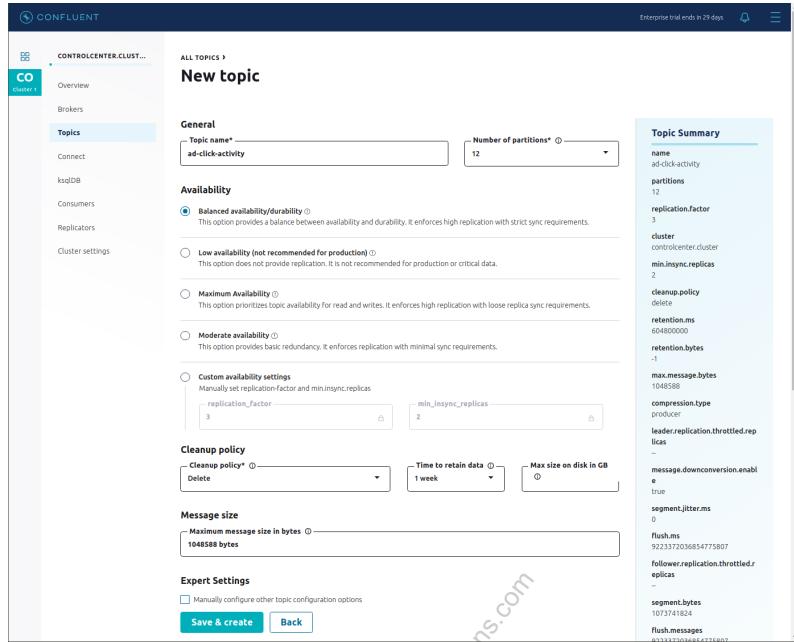
- For dynamic settings that apply to a single broker:

```
kafka-configs --bootstrap-server <broker>:<port> --broker <broker id>  
--describe
```

Do not assume that a configuration will have the same name at the Broker and Topic levels. Often the property names are different even though they configure the same setting.

For example, `message.max.bytes` (Broker) and `max.message.bytes` (Topic) control the largest message size that can be sent to a Topic.

# Setting Topic Configurations with Control Center



The ability to manage Topics through C3 was added in CP 5.0.

# Setting Topic Configurations from the CLI (1)

- Set a Topic configuration at time of Topic creation

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --create \
  --topic my_topic \
  --partitions 1 \
  --replication-factor 3 \
  --config segment.bytes=1000000
```

tejaswin.renugunta@walgreens.com

# Setting Topic Configurations from the CLI (2)

- Change a Topic configuration for an existing Topic

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --add-config segment.bytes=1000000
```

- Delete a Topic configuration

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --delete-config segment.bytes
```

"Delete a Topic configuration" just means that you are resetting it to the cluster default.

tejaswin.renugunta@walgreens.com

# Viewing Topic Settings in Confluent Control Center

The screenshot shows the Confluent Control Center interface. On the left, there's a sidebar with a 'Cluster 1' section containing icons for Overview, Brokers, Topics (which is selected), Connect, ksqlDB, Consumers, Replicators, and Cluster settings. The main area is titled 'ad-click-activity' under 'ALL TOPICS'. It has tabs for Overview, Messages, Schema, and Configuration, with Configuration selected. Below the tabs is a table of topic configurations:

name	ad-click-activity	
partitions	12	
min.insync.replicas	2	
cleanup.policy	delete	
retention.ms	604800000	
max.message.bytes	1048588	
retention.bytes	-1	

At the bottom are two buttons: 'Edit settings' and 'Show full config'.

# Viewing Topic Settings from the CLI

- Show the Topic configuration settings

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --describe \
  --topic my_topic
Configs for topic 'my_topic' are segment.bytes=1000000
```

- Show the Partition, leader, replica, ISR information

```
$ kafka-topics \
  --bootstrap-server broker_host:9092 \
  --describe \
  --topic my_topic
Topic:my_topic PartitionCount:1      ReplicationFactor:3
Configs:segment.bytes=1000000
  Topic: my_topic Partition: 0      Leader: 101 Replicas: 101,102,103    Isr:
101,102,103
```

Note that only non-defaulted values are displayed. You will need to check the configurations page (<https://docs.confluent.io/current/installation/configuration/broker-configs.html>) to see defaults. If the custom defaults were set for your cluster, you can view the Broker properties by looking at the start of the `/var/log/kafka/server.log` file.

You can use the `--help` option with the `kafka-topics` and `kafka-configs` commands to see the options that are available. Both of these commands can be used to modify running Topics without a restart.

# Inspecting Topics

The screenshot shows the Confluent Control Center web interface. On the left, there's a sidebar with 'Overview', 'Brokers', and 'Topics' sections. Under 'Topics', 'ad-click-activity' is selected. The main area has tabs for 'Overview', 'Messages' (which is active), 'Schema', and 'Configuration'. Below these tabs is a search bar with 'Filter by keyword' and 'Jump to offset' dropdowns, along with 'Query in KSQL' and 'Newest' buttons. The 'Messages' section lists three messages:

- Best kayaking rivers in west Virginia  
Partition: 0 Offset: 0 Timestamp: 1591384263849
- Best trails in the Virginia  
Partition: 2 Offset: 0 Timestamp: 1591384246993
- Best beaches on the east coast  
Partition: 7 Offset: 0 Timestamp: 1591384212626

- Optional feature that is enabled at the application level (Control Center settings)
- Allows viewing of key and value of messages
  - Does not deserialize headers
  - Deserializes **AVRO**, **Protobuf**, and **JSON** formats for Topics using Schema Registry
  - Only deserializes **String** format for Topics not using Schema Registry

---

The feature is enabled by default. To disable, set in the C3 configuration file:

```
confluent.controlcenter.topic.inspection.enable=false
```

Topic inspection is introduced in CP 5.0.

# Deleting Topics

- Topic deletion is enabled by default on Brokers in the `server.properties` file
  - `delete.topic.enable` (Default: true)
- Caveats
  - Stop all Producers/Consumers before deleting
  - All Brokers must be running for the `delete` to be successful

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --delete \
  --topic my_topic
```

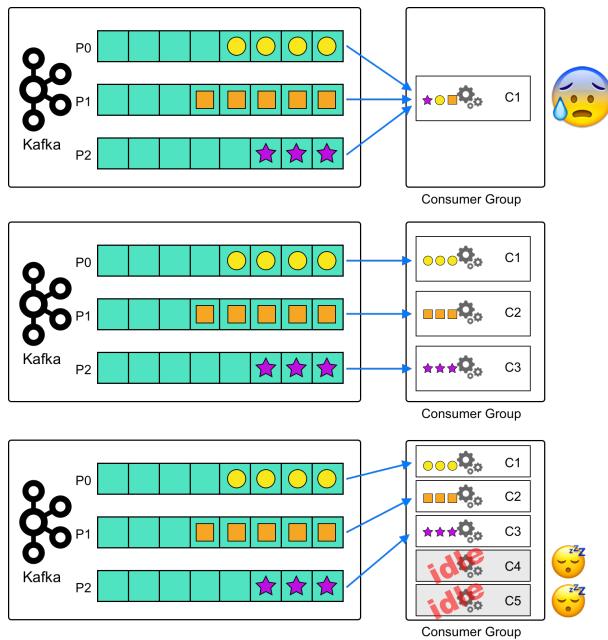
---

By default, Kafka clusters allow auto Topic creation. This will require that all clients are stopped prior to deleting a Topic—otherwise the Topic will be recreated as soon as a client tries to access it.

When a Topic is deleted, the entries in the offsets Topic that are associated with that Topic are not immediately removed. We rely on the messages to be garbage collected over time associated with the time retention policy. The issue is, if you recreate the Topic before the retention time has passed and use the same Consumer Group, the offset won't be valid—it will be related to the old messages. So if you delete a Topic, don't recreate it too quickly (`offsets.retention.minutes` defaults to 1 week).

Prior to Kafka 1.0, the default for `delete.topic.enable` was `false`. To enable Topic deletion, the variable needed to be changed in the `server.properties` file on every Broker in the cluster and required a restart.

# Consumer Group Scalability



The number of useful Consumers in a Consumer Group is constrained by the number of Partitions on the Topic. **Example:** If you have a Topic with three partitions, and ten Consumers in a Consumer Group reading that Topic, only three Consumers will receive data. One for each of the three Partitions.

If there are more Consumers than partitions, the additional Consumers will sit idle. The idea of a hot-standby Consumer is not required but can prevent performance differentials during client failures.

# Adding Partitions

- Use the `kafka-topics` command

```
$ kafka-topics \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --partitions 30
```

- Doesn't move data from existing Partitions
- Messages with the same key will no longer be on the same Partition
  - Workaround: Consume from old Topic and produce to a new Topic with the correct number of Partitions

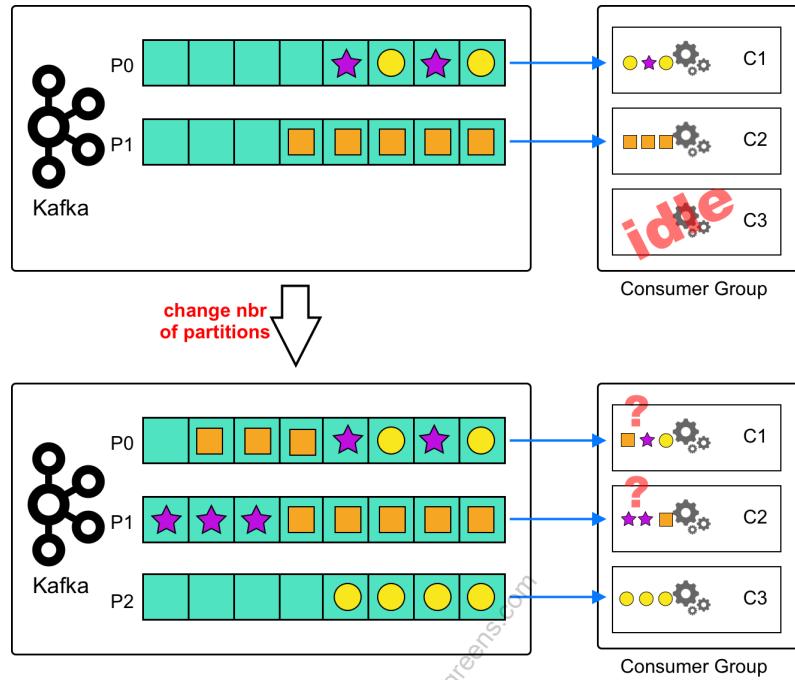
Remember that key-based partitioning uses `hash(key) mod <number-of-partitions>`, so increasing Partition count means what used to go to, e.g., Partition 0 might now go to Partition 15. So messages with a given key will no longer be on a single Partition.

For the workaround (new Topic, copy data), you will have to update all of the clients. Since you will be copying data from the original Topic into a new (larger) Topic in the same cluster, the Topics cannot have the same name. There currently is not a way to rename a Topic.



Kafka does not support reducing the number of Partitions in a Topic. Partitions are append-only event logs, so there is no way for Partition count to be reduced (merging event logs would violate their append-only nature).

# Consumer Groups: Caution When Changing Partitions



On this slide icons with same shapes represent records with same key.

Recall that semantic partition works on the idea that a message will be sent to the partition determined by the formula  $\text{hash}(\text{key}) \% n$ , where  $n$  is the number of partitions. Changing the  $n$  number could change the output of the formula, resulting in messages with the same key being sent to different partitions. This would defeat the purpose of semantic partitioning.

Example: Using Kafka's default Partitioner, Messages with key **K1** were previously written to Partition 2 of a Topic. After repartitioning, new messages with key **K1** may now go to a different Partition. Therefore, the Consumer which was reading from Partition 2 may not get those new messages, as they may be read by a new Consumer

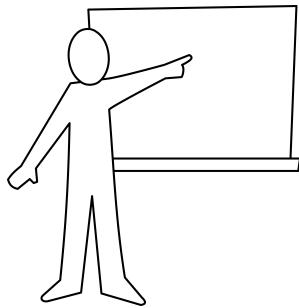
There are strategies to mitigate this problem (e.g., migrating to a larger Topic rather than just expanding the existing Topic) but the best option is to plan appropriately so that you do not have to resize your Topic. Selecting an appropriate number of partitions for your Topic is something that will be discussed later in the course.



In the lower part of the slide the intent to show is that the partitions may after the change contain values with the keys that were assigned prior to the change and new key values, after the change. The downstream Consumers might be confused by that...

tejaswin.renugunta@walgreens.com

# Module Map



- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration**
- Hands-on Lab: **Automating Configuration**
- Monitoring Kafka
- Managing Topic Configurations
- Hands-on Lab: **Increasing Replication Factor ...** ←
- Log Retention and Compaction
- An Elastic Cluster
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

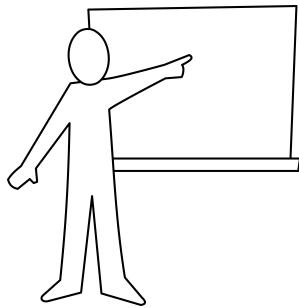
# Hands-On Lab

- In this Hands-On Exercise, you will use the **kafka-reassign-partitions** tool to increase the replication factor of an existing Topic.
- Please refer to **Lab 04 Managing a Kafka Cluster** in the Exercise Book:
  - c. **Increasing Replication Factor**



tejaswin.renugunta@walgreens.com

# Module Map



- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration**
- Hands-on Lab: **Automating Configuration**
- Monitoring Kafka
- Managing Topic Configurations
- Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction ... ←
- An Elastic Cluster
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Managing Log File Growth



- `log.cleanup.policy`
  - `delete`
  - `compact`
  - both: `delete, compact`



Cleanup policy can also be set on a per-Topic basis

Logs are not deleted on consumption because many Consumers might read from a Topic at different times. Instead, the Brokers use a **cleanup** policy to decide when to delete messages. This can be set at the Broker or Topic level.

The cleanup policy is set to `delete` by default. With the `delete` policy, log segment files are deleted when they get older than a given age or if the entire Partition exceeds a given size.

The `compact` policy is used for keyed messages. Only the freshest message with a given key will be kept. For example, if these messages flow into Kafka:

```
{"pie":"the pie is hot"},  
 {"pie":"the pie is warm"},  
 {"pie":"the pie is cold"}
```

then the Broker will periodically compact the log so that only `{"pie":"the pie is cold"}` remains. Log compaction will be discussed in further detail in a later slide.

Combining `delete` and `compact` is useful for Topics that need to be compacted by key but also want keys that are stale (i.e., haven't been updated for some time) to be automatically expired. Sample use cases:

- Order Management: An e-tailer is using the order number as the key to track the state of an order ("101":"placed", "101":"processing", "101":"shipped", "101":"delivered"). Once the package is delivered, the key is never used again and so will stay in the compacted Topic until the retention time has passed.
- A windowed join in Kafka Streams: If using windows of time with many versions of a keyed message, you may only want to retain the latest version of the key message during the window. However, once the window has expired you would like to have the segments

for the window deleted.



Regardless of cleanup policy chosen, cleanup does not delete messages in the currently active segment file (i.e., the segment file which is actively receiving writes from the Producers).

tejaswin.renugunta@walgreens.com

# The Delete Retention Policy

How log segments roll:	When segments are checked:	What the log cleaner checks:
<ul style="list-style-type: none"><li>• <code>log.roll.ms</code></li><li>• <code>log.segment.bytes</code></li><li>• <code>log.retention.ms</code></li></ul>	<ul style="list-style-type: none"><li>• <code>log.retention.check.interval.ms</code><ul style="list-style-type: none"><li>◦ Default: 5 min</li></ul></li></ul>	<ul style="list-style-type: none"><li>• <code>log.retention.ms</code></li><li>• <code>log.retention.bytes</code> (disabled by default)</li></ul>
<ul style="list-style-type: none"><li>• Segments whose newest message is older than <code>log.retention.ms</code> will be deleted</li></ul>		

Recall that segment files are "rolled" on a regular basis; when a segment file reaches a specified age or size, it is closed and a new segment file is created to receive new data. Settings that affect when a new segment file is rolled out are:

- `log.roll.ms` (default 1 week)
  - limit on how long a segment file is active before a new one is rolled.
- `log.segment.bytes` (default 1 GB)
  - limit on how large the segment file can become before a new one is rolled.
  - If a record being added to a segment file will cause it to exceed the `log.segment.bytes` value, that segment file will be closed, a new segment file added, and the record will be written to the new segment.
- The time based log retention setting that is in effect. The default setting is `log.retention.hours=168` (7 days) but this is superceded if `log.retention.minutes` or `log.retention.ms` is set. The default value for these two settings is `null`.
  - If the active segment is older than this amount, then it will be closed and a new segment file rolled out.
  - This is an edge case since segments usually roll much more frequently than this amount of time. This will be discussed further in an upcoming slide.

The log cleaning process will trigger every `log.retention.check.interval.ms` (default 5 minutes). When the cleanup policy contains `delete`, the log cleaner will delete inactive segments according to age or, less commonly, Partition size. Once the log cleaner thread is triggered, it will check inactive log segment files and delete the files that don't pass its checks:

- Age: If the newest message in a segment file is older than `log.retention.ms` (default 1 week), it will be deleted

- Size: If the entire Partition exceeds `log.retention.bytes`, then when the log cleaner is triggered, segment files will be deleted from oldest to newest until the size of the Partition is back within `log.retention.bytes`
  - The default for `log.retention.bytes` is -1, meaning this function is disabled.
  - Another edge case occurs where the active segment is larger than `log.retention.bytes`. In this case, the active log segment will be deleted. Temporarily setting a Topic's `retention.bytes` to 0 can therefore be used to delete the data in the Topic. This will be shown in an upcoming slide.

Note that the cleanup policy applies to whole log segment files, not individual messages. The age of a segment file is determined by the timestamp of its newest message. This means that messages are guaranteed to live *at least* as long as the retention time, but many messages in the segment file will remain for longer than the retention time.

tejaswin.renugunta@walgreens.com

# Deleting All Messages in a Topic (1)

1. Turn off all Producers and Consumers
2. Temporarily configure `retention.ms` to 0

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --add-config retention.ms=0
```

3. Wait for cleanup thread to run (every 5 minutes by default)

tejaswin.renugunta@walgreens.com

# Deleting All Messages in a Topic (2)

## 4. Restore default `retention.ms` configuration

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --delete-config retention.ms
```



Do not just delete log files!

The method here assumes the Topic's `cleanup.policy` is set to `delete`.

This idea relates to the aforementioned edge case with `retention.ms`. If an active segment is older than `retention.ms`, then it is closed and a new segment file is rolled. Setting `retention.ms=0` immediately closes the active segment. Now, once the log cleaner check interval passes, the log cleaner will delete all segments older than 0 ms (i.e. all segments). In virtually all practical situations, `retention.ms` is much longer than the time it takes for a log to roll to the next segment, which is why this idea can be counterintuitive at first.

There is another way to accomplish this task that is beyond the scope of this course. Developers can use the AdminClient API in their client code (Producer or Consumer) to delete all records prior to a specified offset.



This is not recommended for production systems

# Compact Policy Use Case

Keep only the **freshest** value of each key

- Examples:
  - Database change capture
  - Real-time table lookups during stream processing
  - Event sourcing



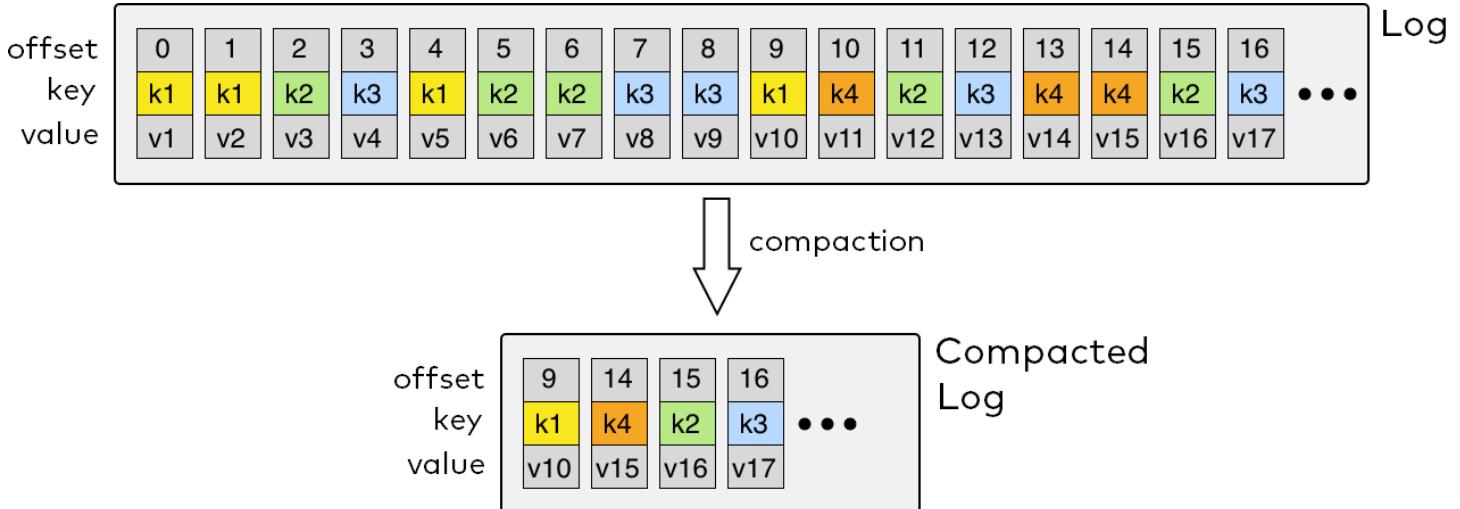
The **compact** cleanup policy is specific to keyed messages. It is designed for applications that require only the freshest value of any given key, for example to help in-memory applications recover from failure with the latest state.

Sample use cases:

- Database change capture: maintain a replica of the data stream by key (e.g., a search index receiving real-time updates but needing only the most recent entry)
- Stateful stream processing: journaling arbitrary processing for high availability (e.g. Kafka Streams or anything with "group by"-like processing)
- Event sourcing: co-locates query processing with application design and uses a log of changes as the primary store for the application

With keyed messages, all message of a given key land on the same Partition unless a custom partitioning strategy is used. The **compact** policy only guarantees the freshest value of a given key on a per-Partition basis; there still may be multiple messages with the same key if the messages are on different Partitions.

# Log Compaction: What is it?

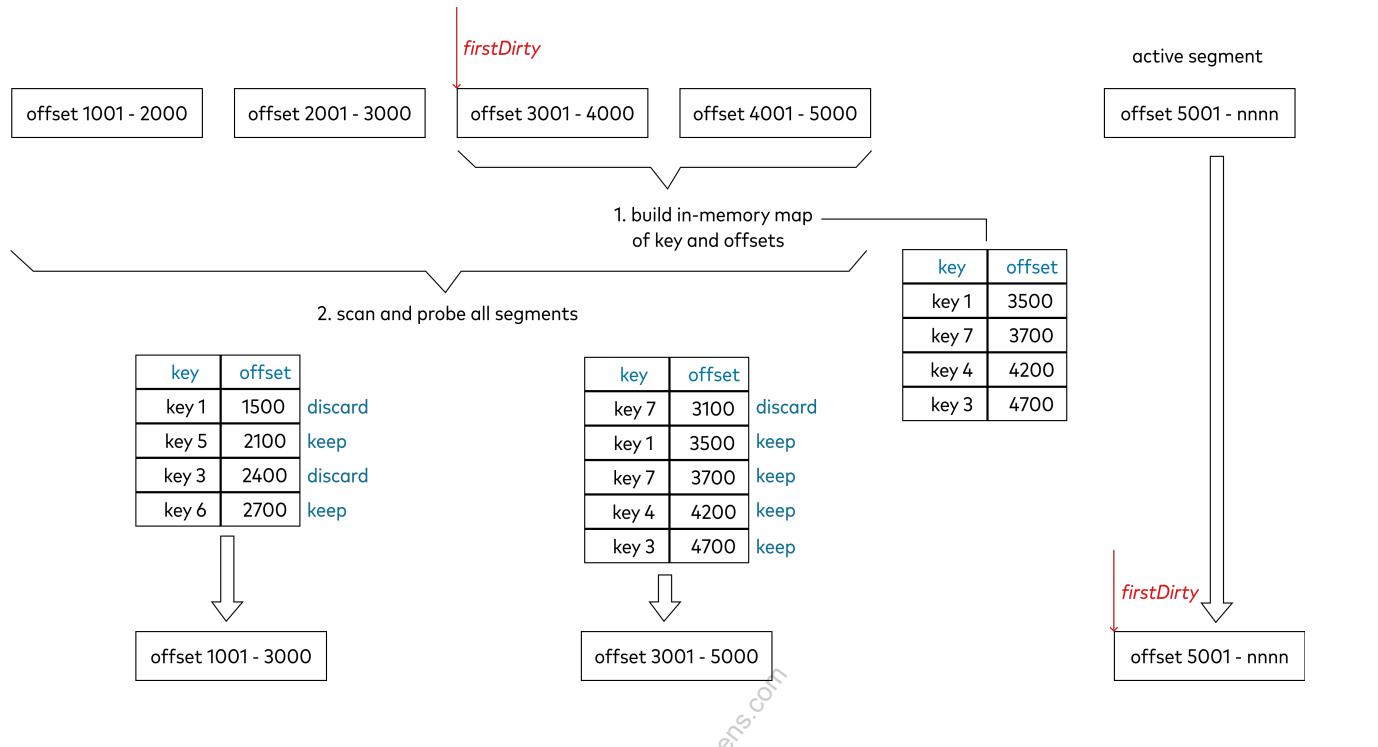


The **compact** policy only works for keyed messages. The log on top contains all events/records that are produced by the data source (each event in this slide has an offset or time, a key, and a value). Events with the same key are color coded with the same color for easier readability.

Some of the messages have the same key. When setting a Topic's **cleanup.policy** to **compact**, the log cleaner will only preserve the last message with that key; older messages with that key will be deleted. This is called "log compaction."

Compacted logs are useful for restoring state after a crash or system failure. Without compaction, a Consumer can always read the entire log and eventually reach the most up-to-date state for each key, but log compaction allows the Consumer to get to the end state faster since it doesn't read stale data.

# Log Compaction: Implementation



The top line of the diagram represents log segments, arranged from oldest to newest. Everything to the right of the "firstDirty" marker is considered dirty. Dirty segments are used to build an in-memory map of the newest instance of each key and its offset. The log cleaner then makes another scan of all clean and dirty segments. Each message is checked to see if its key has a match in the in-memory map.

- If a match is not found, that message is the latest version of that key so it is preserved in a new segment file.
- If a match is found and its offset is different from the matching record in the in-memory map, then the map contains the latest version of the key and the old message is dropped.
- If a match is found and its offset is the same as the matching record in the in-memory map, the message is preserved in a new segment file.

This process will generate a new set of segment files which only preserve the messages with the freshest values, so the new segment files will be smaller and can be combined if necessary. Once the scan finishes, the old segment files are replaced by the new segment files, and the "firstDirty" pointer is moved.

# Log Messages During Cleaning

When the cleaning process is taking place, you will see log messages like this:

```
Beginning cleaning of log my_topic,0  
Building offset map for my_topic,0 ...  
Log cleaner thread 0 cleaned log my_topic,0 (dirty section=[100111,200011])
```

Switching to the new segment files can be done quickly. Each Broker retains an in-memory list of which segment files the Broker contains. After switching to the new segment files, new fetch requests will see the new segment file list. What if there are outstanding fetch requests working from the old segment files that have not been finished? Older segments are preserved for an amount of time (1 minute, by default) before they are deleted to allow the outstanding fetch requests to finish. If they do not finish in this time, the Consumer will get an error.

What if a Consumer Group pauses and comes back but the offset it is supposed to read from has been removed by a clean-up policy? The Broker will advance to the next highest offset which follows the one requested.

tejaswin.renugunta@valgusens.com

# Log Compaction: Important Configuration Values

- `log.cleaner.min.cleanable.ratio` (default 0.5)
    - Triggers log clean if the ratio of `dirty/total` is larger than this value
  - `log.cleaner.io.max.bytes.per.second` (default infinite)
    - throttle log cleaning
- 

Common log cleaner configurations to tune:

- `log.cleaner.min.cleanable.ratio`: trigger the log cleaner when the percentage of dirty data exceeds this value. Defaults to 50%.
- `log.cleaner.io.max.bytes.per.second`: throttles the amount of system resources that the cleaner can use. Due to the amount of reads and writes, cleaning is I/O intensive. Default is infinite.

Adjust these properties carefully—more frequent log cleaning means more I/O time, higher disk utilization.

For Partitions with "high cardinality" (many unique keys), the cleaner threads may take a very long time to process a log compared to other logs with "low cardinality". Situations like this may require an increase to the `log.cleaner.dedupe.buffer.size` or `log.cleaner.threads` settings.

Here is some additional information about tuning log cleaning: The value of `log.cleaner.threads` depends on whether the cleaner job is CPU bound or IO bound. In general, it's probably IO bound. So this can be set to the number of disks per Broker. However, if compression is enabled, CPU could be the bottleneck. In this case, it can be set up to the number of cores on the Broker. In either case, it may be useful to set `log.cleaner.io.max.bytes.per.second` to avoid the cleaner consuming too much resources. For `log.cleaner.dedupe.buffer.size`, in general, the higher it is, the fewer rounds of cleaning are needed. If one knows the # of unique keys in a log, one can set it to `#keys * 24 bytes` but bound it by the largest amount of memory one can afford in the Broker.

You can monitor disk utilization with commands like `iostat`.

# Deleting Keys with Log Compaction



- Tombstone Messages:
  - delete key `K` by sending `{K:null}`
  - Consumer has `log.cleaner.delete.retention.ms` time to consume `K` before it is deleted (default 1 day)

As keys are retired, many systems will send delete messages. The simplest approach would be to retain delete messages forever. But since the purpose of deletes is typically to free up space, this approach would have the problem that the Commit Logs would end up growing forever if the keyspace keeps expanding and the delete markers consume some space. Tombstones (the Kafka implementation of a delete message) are keyed messages with a null value.

However, a delete message should not be removed too quickly or it can result in inconsistency for any Consumer of the data reading the tail of the log. Consider the case where there is a message with key `K` and a subsequent delete for key `K`. If log compaction removes delete messages, there is a race condition between a Consumer of the log and the log compaction action. Once the Consumer has seen the original message, we need to ensure it also sees the delete message; this might not happen if the delete message happens too quickly. As a result, the Topic can be configured with a configurable SLA for delete retention (`delete.retention.ms`). This SLA is in terms of time from the last cleaning. A Consumer that starts from the beginning of the log must consume the tail of the log in this time period to ensure that it sees all delete messages.

If an application needs to be able to send null values that will not be mistaken as tombstones, you need to introduce a null-type – like a `NullInteger` – so it looks like a regular message with non-null value to Kafka.

# Monitoring Log Compaction

- JMX metrics:

```
kafka.log:type=LogCleanerManager,name=max-dirty-percent
```

```
kafka.log:type=LogCleaner,name=cleaner-recopy-percent
```

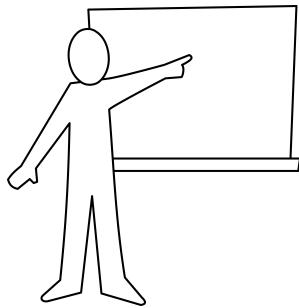
```
kafka.log:type=LogCleaner,name=max-clean-time-secs
```

---

In particular, watch **cleaner-recopy-percent** and **max-clean-time-secs** on compacted Topics. High values for either or both of these could indicate a high number of stale keys that are not being updated. Consider using tombstones or the delete-compact combination cleanup policy to get rid of the stale keys.

tejaswin.renugunta@walgreens.com

# Module Map

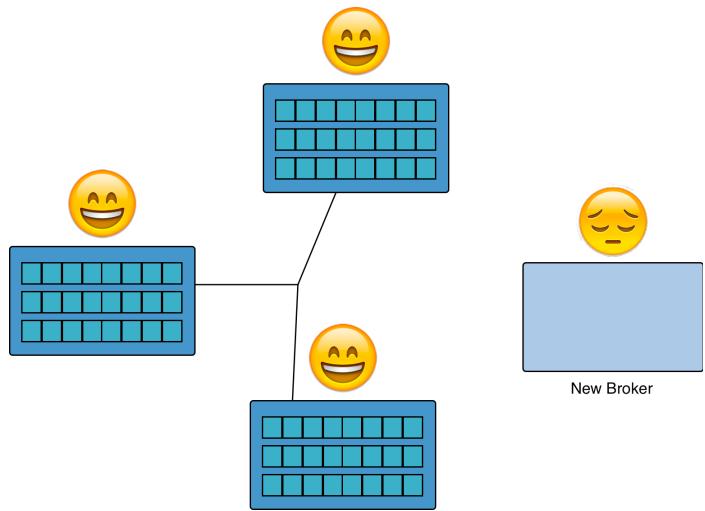


- Installing and Running Kafka
- Hands-on Lab: **Exploring Configuration**
- Hands-on Lab: **Automating Configuration**
- Monitoring Kafka
- Managing Topic Configurations
- Hands-on Lab: **Increasing Replication Factor**
- Log Retention and Compaction
- An Elastic Cluster ...
- Hands-on Lab: **Kafka Administrative Tools**

tejaswin.renugunta@walgreens.com

# Expanding the Cluster

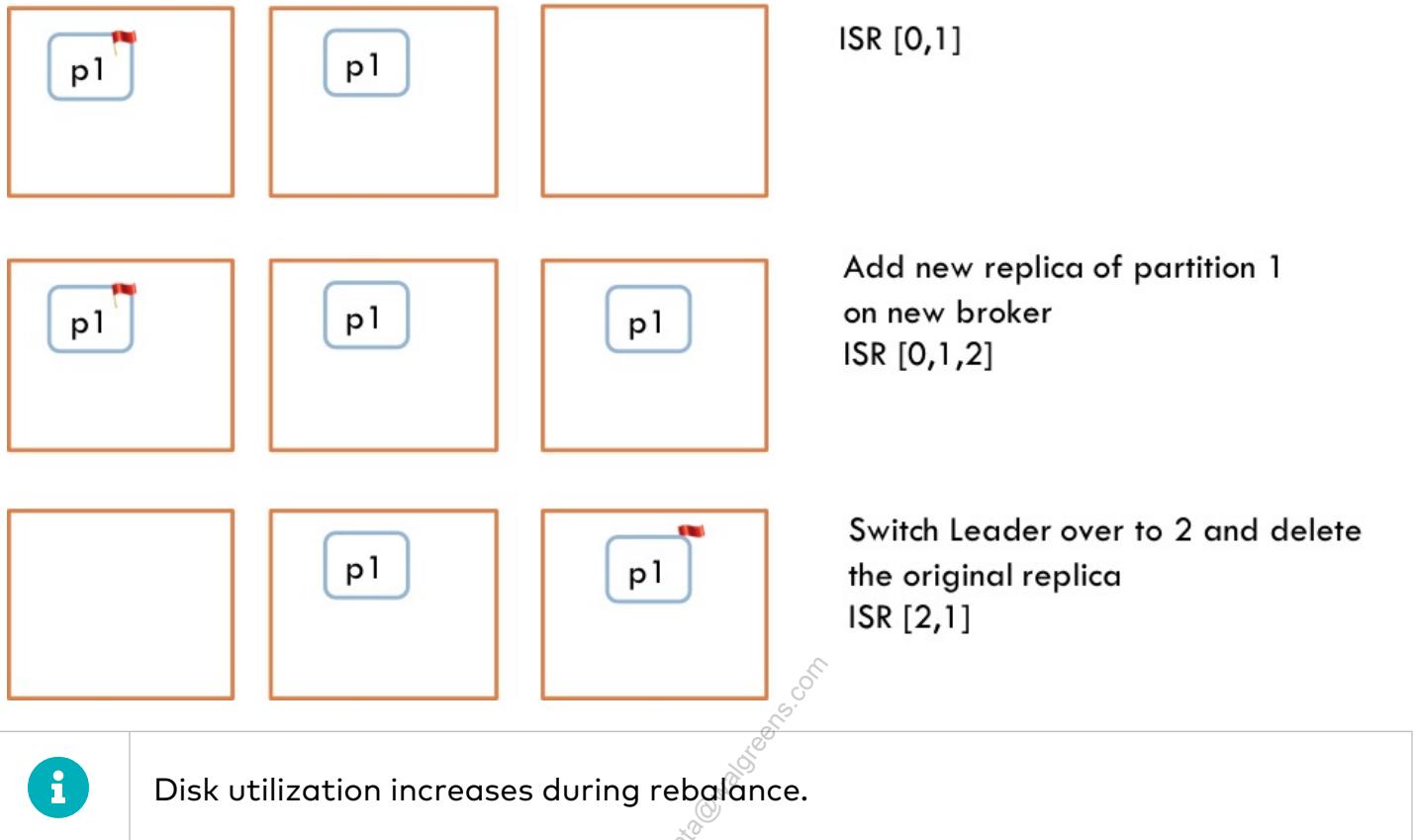
- Kafka doesn't automatically move data to new Brokers
  - Unbalanced leaders
  - Unbalanced disk utilization
- How to balance with zero downtime?
  - Confluent's **Auto Data Balancer**
  - Built-in `kafka-reassign-partitions` tool



Often, as Kafka becomes more integral to environments, the demand will exceed the capabilities of your initial cluster. Adding capacity to a Kafka Cluster is easy - add a new Broker with a unique Broker ID and point it at the same ZooKeeper ensemble.

However, adding Brokers to the Cluster does not automatically move Partitions. Partition migration creates significant load on the networks so this type of action only takes place when requested by an administrator. However, when it does happen, Partition migration can be performed while the Cluster is running - no downtime is required.

# Partition Movement with Auto Data Balancer



In order to move the Partitions with no downtime, Auto Data Balancer (ADB) leverages the replication feature. The new Broker location is added as an additional Partition temporarily. Once the Partitions are replicated to all of the final Broker locations, the Partitions can be removed from their original locations.

Once a Partition migration has started, it must complete. There is no way to cancel the operation.

# Disk Utilization During Rebalancing (1)

```
$ confluent-rebalancer execute \
--bootstrap-server broker_host:9092 \
--metrics-bootstrap-server kafka-1:9092,kafka-2:9092 \
--throttle 1000000 \
--verbose \
--force
```



Auto Data Balancer is part of Confluent Enterprise.

The slide shows an example of running Confluent Auto Data Balancer. The next slide shows the result of running the command.

The `--throttle` parameter is used to specify the maximum bandwidth, in bytes per second, allocated to moving replicas.

# Disk Utilization During Rebalancing (2)

```
Computing the rebalance plan (this may take a while) ...
You are about to move 17 replica(s) for 14 partitions to 4 broker(s) with total size 827.2 MB.
The preferred leader for 14 partition(s) will be changed.
In total, the assignment for 15 partitions will be changed.
The minimum free volume space is set to 20.0%.
The following brokers will have less than 40% of free volume space during the rebalance:
Broker    Current Size (MB)  Size During Rebalance (MB)  Free % During Rebalance  Size After Rebalance (MB)  Free % After Rebalance
 0        413.6              620.4                  30.1                519.6                  30.5
 2        620.4              723.8                  30.1                520.8                  30.5
 3        0                  517                   30.1                520.8                  30.5
 1        1,034              1,034                 30.1                519.6                  30.5

Min/max stats for brokers (before -> after):
Type   Leader Count      Replica Count      Size (MB)
Min   12 (id: 3) -> 17 (id: 0)  37 (id: 3) -> 43 (id: 3)  0 (id: 3) -> 517 (id: 1)
Max   21 (id: 0) -> 17 (id: 0)  51 (id: 1) -> 45 (id: 0)  1,034 (id: 1) -> 517 (id: 3)

No racks are defined.

Broker stats (before -> after):
Broker    Leader Count      Replica Count      Size (MB)      Free Space (%)
 0        21 -> 17          48 -> 45          413.6 -> 517  30.5 -> 30.5
 1        20 -> 17          51 -> 44          1,034 -> 517  30.5 -> 30.5
 2        15 -> 17          40 -> 44          620.4 -> 517  30.5 -> 30.5
 3        12 -> 17          37 -> 43          0 -> 517       30.5 -> 30.5

Would you like to continue? (y/n):
Rebalance started, its status can be checked via the status command.

Warning: You must run the status or finish command periodically, until the rebalance completes, to ensure the throttle is removed. You can also alter the throttle by re-running the execute command passing a new value.
```

Auto Data Balancer shows disk utilization during and after rebalance. In order for ADB to compute the rebalance plan, it needs to know the size of each Partition in the Kafka cluster. This data is published by the ConfluentMetricsReporter to a configurable Kafka Topic (`_confluent-metrics` by default)

# Rebalancing Caveats

- No way to cancel
- Only one rebalance active at a time
- Resource-intensive
  - Throttle the rebalance:

```
$ confluent-rebalancer execute \
  --bootstrap-server broker_host:9092 \
  --metrics-bootstrap-server kafka-1:9092 \
  --throttle 50000000
```



Only attempt rebalance when all Brokers are live. Otherwise, rebalance will not complete until all Brokers are live.

Moving replicas can be a very resource-intensive process, creating an unbounded load on inter-cluster traffic. This affects clients interacting with the cluster when a data movement occurs. Throttling mechanisms for leaders and followers can be configured when running Auto Data Balancer or [kafka-reassign-partitions](#).

In order for ADB to complete successfully, all steps must complete - including deleting the old replicas. If the Broker that was the original location of the replica is down, the migration can't complete until the old Broker comes back up with the old replica and that replica can be deleted.

# Auto Data Balancer vs kafka-reassign-partitions

Feature	Auto Data Balancer	kafka-reassign-partitions
Automated balancing cluster-wide, no math required	Yes	No, per Topic
Faster rebalancing with optimized Partition movement	Yes	No, and risk running out of interim disk space
Monitoring and statistics	Yes, per-Broker statistics	No
Decommission Brokers	Yes, automated with <code>--remove-broker-ids</code> option	Yes, manual reassignment
Increase replication factor of a Topic	No	Yes, with some manual work
Balance Partitions across log directories	No	Yes

This section demonstrated Partition migration using the Auto Data Balancer (ADB) tool available through Confluent Enterprise. Kafka Core includes a tool called `kafka-reassign-partitions`. If students are interested in this tool, refer them to the lab exercise entitled "Appendix: Reassigning Partitions in a Topic - Alternative Method".

ADB gives the ability to automatically balance a cluster, but `kafka-reassign-partitions` gives granular control over the number and placement of Partitions. For example, `kafka-reassign-partitions` can increase a Topic's replication factor and even move Partitions to specific log directories (for example, to load balance Partitions across disks). For example, to increase the number of replicas of Partition 3 of Topic `my-topic` from 1 to 2 and ensure one replica lands in the `/var/lib/kafka/data-3` log directory of Broker 101, create a `json` file (here called `reassign.json`):

```
{  
  "partitions": [  
    {  
      "topic": "my-topic",  
      "partition": 3,  
      "replicas": [101, 103],  
      "log_dirs": ["/var/lib/kafka/data-3", "any"]  
    }  
  ],  
  "version": 1  
}
```

Then run `kafka-reassign-partitions` using the `reassignment-json-file` and `execute` options:

```
$ kafka-reassign-partitions \  
  --bootstrap-server kafka-1:9092 \  
  --bootstrap-server broker_host:9092 \  
  --reassignment-json-file reassign.json \  
  --execute
```



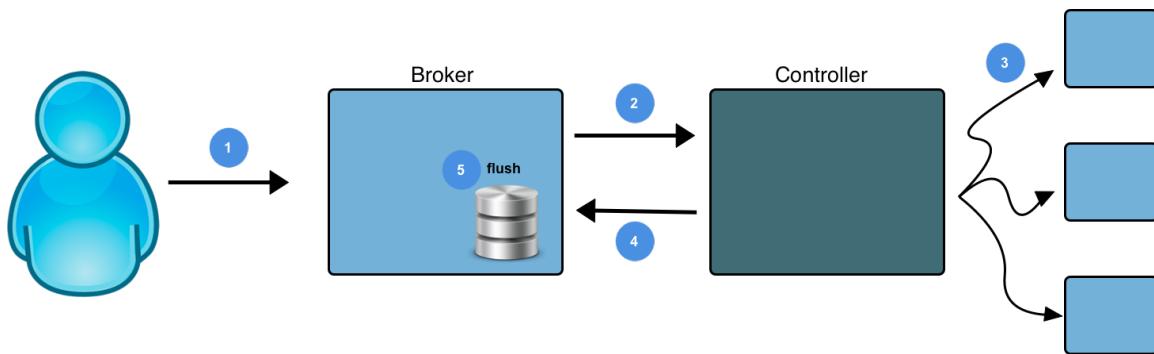
The `--bootstrap-server` option must be specified if designating specific log directories.

# Shrinking the Cluster

- Why reduce the number of Brokers in the cluster?
    - Maintenance on a Broker
    - Reduce cost during periods of low cluster utilization
  - Decommissioning a Broker
    1. Use Auto Data Balancer or `kafka-reassign-partitions` to reassign its Partitions to other Brokers
    2. Perform a controlled shutdown
- 

Why shrink a cluster? An easy example is an online retailer hosting their Kafka cluster in a cloud-based environment. During certain times of the year, they expect to handle larger volumes of transactions. Spreading Partitions across more Brokers can increase available throughput and enable them to handle the increased traffic. However, once traffic returns to normal levels, it does not make sense to pay the cloud vendor for the extra capacity. ADB can be used to decommission Brokers easily with the `--remove-broker-ids` option. The `kafka-reassign-partitions` tool, however, requires manually moving Partitions from Brokers that are planned for decommission.

# Controlled Shutdown



1. Administrator sends a **SIGTERM** to the Broker Java process, e.g. `kafka-server-stop`
2. The Broker sends request to Controller.
3. The Controller facilitates leader elections
4. Controller **acks** Broker.
5. Broker flushes file system caches to disk.
6. Broker shuts down.

Relevant configuration settings:

- `controlled.shutdown.enable` (default `true`)
- `controlled.shutdown.max.retries` (default 3)
- `controlled.shutdown.retry.backoff.ms` (default 5 sec)

The sequence of events during a controlled shutdown is:

1. Administrator sends a **SIGTERM** to the Broker Java process, e.g. `kill -SIGTERM <pid>`
2. The Broker sends a request to the Controller to tell the Controller it's planning to shut down.
3. The Controller will determine which Partitions the Broker is a leader for and does a leader election for each of these Partitions to move to the leader role to another Broker.
4. Once the elections are complete, the Controller sends an acknowledgment back to the Broker.
5. The Broker explicitly flushes the file system caches to disk.
6. The Broker shuts down.

The key benefit of a controlled shutdown is minimized downtime for the Partitions. When a leader is elected, there is a brief outage for the clients between when the leader goes offline and the client does a metadata update to determine which Broker is the new leader. During a controlled shutdown, the Controller only takes a single Partition offline at a time while that Partition's leader is reelected. For the uncontrolled shutdown (or Broker crash), all Partitions for which the Broker was the leader will go offline at the same time. The Controller is only able to elect one leader at a time. For example, assume the crashed Broker is the leader for 10 Partitions. The tenth Partition (the last one whose leader is elected) will be offline significantly longer than the first Partition. A controlled shutdown would have provided more reasonable outage time for all Partitions.

tejaswin.renugunta@walgreens.com

# Replacing a Failed Broker

- On a new server, start a new Broker with the same value for `broker.id`
- The new server will automatically bootstrap data
- If possible, do the Broker replacement at an off-peak time



Broker will copy the data as fast as it can during recovery. This can have a significant impact on the network.

Kafka does not migrate Partitions if a Broker fails. The procedure to recover a failed Broker is:

1. Replace the hardware
2. Install the OS and Kafka
3. Assign the Broker the identity of the lost system (Broker ID and ZooKeeper hosts to contact)
4. Start the Broker and let the replication features rebuild the Partitions.

Kafka does not care about the host or IP address where it runs - Broker ID is all that identifies the system to the cluster.

Be careful—the Broker will copy the data as fast as it can during recovery and there is no throttling tool. This can have a significant impact on the network. If possible, it is recommended to avoid scenarios where a Broker loses all of its data and has to recover every Partition via replication. For example, specify many log directories with `log.dirs` and mount a separate disk to each log directory. That way, a disk failure means only some of a Broker's Partitions need to be recovered via replication over the network. If in a public cloud environment, another example would be to mount a distributed file system to the log directory specified with `log.dirs` (e.g. AWS Elastic Block Storage). These systems have their own guarantees about preventing data loss, so Brokers can come back online and only recover a much smaller amount of data (only whatever was lost in the page cache and whatever data has been produced while the Broker was down). These considerations will be given more detail in a later module.

Throttling can be achieved during Broker recovery using `kafka-configs`, but this requires you to figure out which Partitions belong on the failed Broker. There is no tool for this as of Kafka 2.2, but it could be accomplished by running `kafka-topics --describe` for each Topic in the cluster and parsing (e.g. with `awk`) to find all the Topic-Partitions that belong to

a specific Broker. See <https://cwiki.apache.org/confluence/display/KAFKA/KIP-73+Replication+Quotas> for more details.

tejaswin.renugunta@walgreens.com

# Hands-On Lab

- In this Hands-On Exercise you will delete a Topic, reassign Partitions, and simulate a completely failed Broker
- Please refer to **Lab 04 Managing a Kafka Cluster** in the Exercise Book:
  - d. **Kafka Administrative Tools**



tejaswin.renugunta@walgreens.com

# Module Review



- Clusters can be installed and configured automatically
- Kafka provides command line utilities to modify Broker and Topic configurations
- Monitor a Kafka cluster through log files and metrics
- Log retention policies ensure the log doesn't grow unchecked
- Log compaction is useful if only the freshest value for each key is required
- Kafka clusters can expand, shrink, and recover from failure
- Kafka provides utilities for moving Partitions to balance the load across the Brokers

tejaswin.renugunta@walgreens.com

# 05: Optimizing Kafka's Performance



CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Providing Durability
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance ... ↩
6. Kafka Security
7. Data Pipelines with Kafka Connect
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

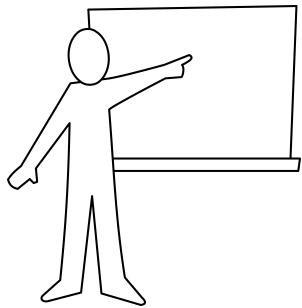


After this module you will be able to:

- Explain why batching helps performance
- Explain the relationship between Partitions and throughput
- Describe the anatomy of requests on a Broker
- Explain how Consumer Groups read messages from Kafka
- Performance-tune the cluster

tejaswin.renugunta@walgreens.com

# Module Map



- Terminology Review ... ←
- Producer Performance
- 🌐 Hands-on Lab: **Exploring Producer Performance**
- Broker Performance
- Broker Failures and Recovery Time
- Load Balancing Consumption
- 🌐 Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance
- Performance Testing
- 🌐 Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# The Meaning of Performance

- **Throughput**
  - amount of data moving through Kafka per second
- **Latency**
  - The delay from the time data is written to the time it is read
- **Recovery Time**
  - The time to return to a "good" state after some failure



---

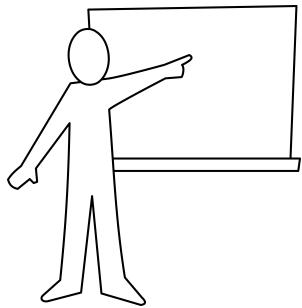
Throughput is often measured in megabytes per second (MBps). Even a small three-Broker cluster with modest hardware can achieve on the order of 10-100 MBps. The more data pushed through the cluster, the better.

Latency is often measured in milliseconds (ms). Depending on how a cluster is tuned, end-to-end latency between when a message is produced to when it is consumed can be anywhere from single digit ms to a few seconds. Interactive and proactive applications require very low latency, while asynchronous applications can tolerate much longer latencies.

Recovery time is important to operations because it has the greatest effect on availability calculations. Availability is the fraction of the time that a service meets its predefined requirements. Availability is typically defined as  $F / (F+R)$  where  $F$  is the mean time to **failure** and  $R$  is the mean time to **recovery**. In this equation, lowering  $R$  by some fixed percentage has a much greater effect on availability than increasing  $F$  by the same percentage.

When tuning a Kafka environment for performance, ensure that you are talking to your customer to understand what is most important to them. Although throughput and latency are not mutually exclusive goals, tuning one will affect the other. Focus on the performance metric that is considered more critical to your customer.

# Module Map



- Terminology Review
- Producer Performance ... 
-  Hands-on Lab: **Exploring Producer Performance**
- Broker Performance
- Broker Failures and Recovery Time
- Load Balancing Consumption
-  Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance
- Performance Testing
-  Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# Batching for Higher Throughput



Imagine there is a line of 50 kids trying to get to school. Think about two strategies for getting them to school:

1. There is a steady flow of SmartCars. The first student enters the SmartCar and goes to school at the speed limit. After that car leaves, there is another SmartCar that the second student takes, and so on.
2. One big bus comes to pick up all the students at once before heading to school at the speed limit.

Here are some questions to consider:

- If you were a student, which option would get you to school the fastest?
  - The SmartCar would be faster for an individual student because they don't have to wait for all students to load before taking off.
- If you were the school administrator, which option would get the whole batch of students to school the fastest?
  - The bus would be faster because each SmartCar has a "preparation time" per student (the car pulls up, the student walks to the door, opens it, gets in, buckles seat belt, etc.), whereas the bus opens its door once and loads everyone more efficiently and then goes.

# Hands-On Lab

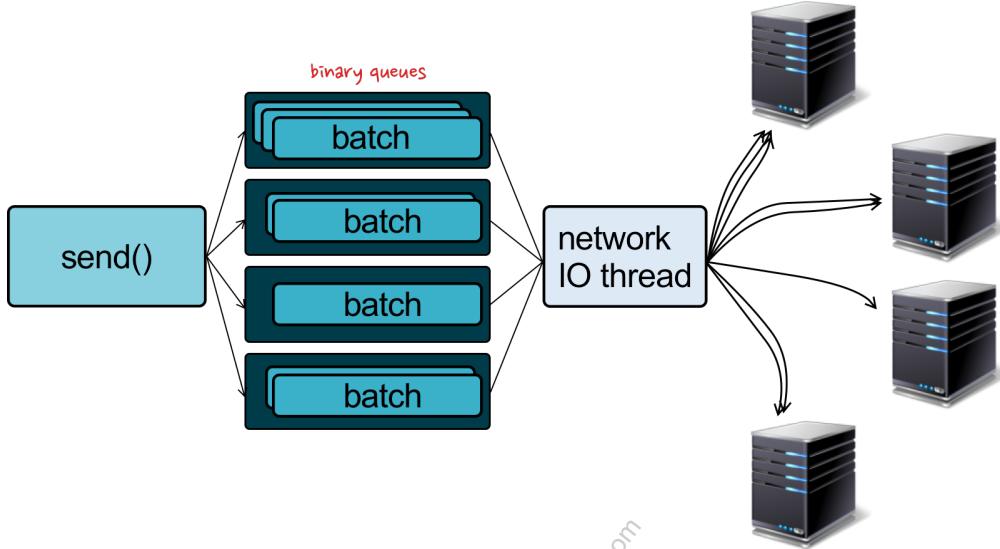
- In this Hands-On Exercise, you will investigate performance tradeoffs of different Producer configurations.
- Please refer to **Lab 05 Optimizing Kafka's Performance** in the Exercise Book:
  - a. **Exploring Producer Performance**



tejaswin.renugunta@walgreens.com

# Producer Architecture

- **Pipelining:** multiple in-flight send requests per Broker
  - `max.in.flight.requests.per.connection` (default: 5)



When a Producer creates a message, pushing that data to the cluster is a two-stage process:

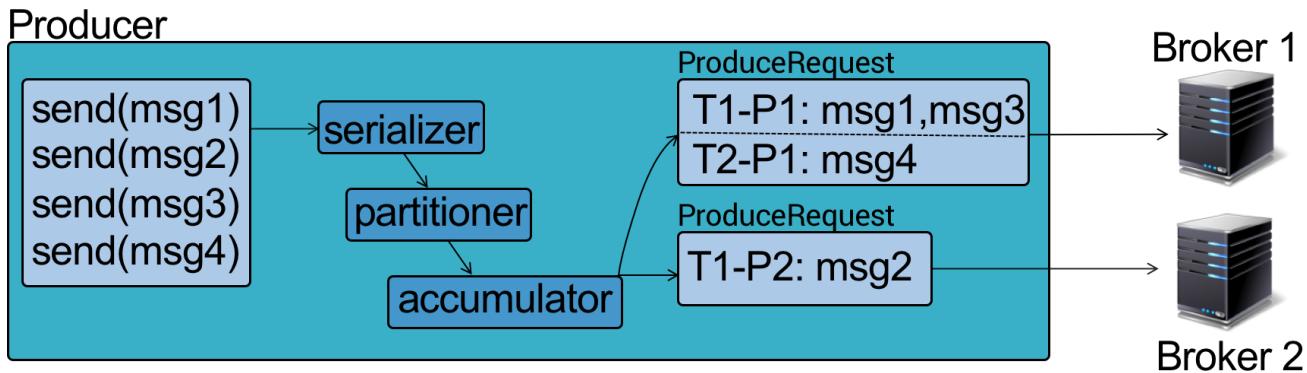
1. A method called `send()` places messages into binary queues in RAM on the Producer system
2. An internal thread will push the messages to the leader Broker for the specific Partitions.

Requests are pipelined, which means there can be multiple in-flight send requests per Broker. The Producer setting that governs this is

`max.in.flight.requests.per.connection` (default: 5), which is set in the Producer code.

But why use a two-stage process to get message into the Brokers? This enables the Producer to send messages in real-time or as batches.

# Batching Messages (1)



1. First, batch messages by Partition
2. Then, collect batches into ProduceRequests to Brokers

Though Kafka is designed to handle messages in real-time, it can also be tuned to use batching for higher throughput at the cost of higher latency. When multiple messages are sent to the same Partition, a Producer may attempt to batch them. If multiple Partitions have leaders on the same Broker, the Producer collects the appropriate batches together into a **ProduceRequest** to that Broker. Batching provides better throughput since grouping together reduces the number of RPCs (remote procedure calls) and so the Brokers have less to process. Batching also typically makes compression more effective.

The arrows pointing to Brokers each represent a **ProduceRequest**. A single **ProduceRequest** can contain one or more batches of records. The **ProduceRequest** includes the Topic and Partition metadata for each **RecordBatch** so that Brokers know exactly how to handle the incoming data.



Not shown is the fact that there is a separate buffer for each Partition as the Producer accumulates messages into batches.

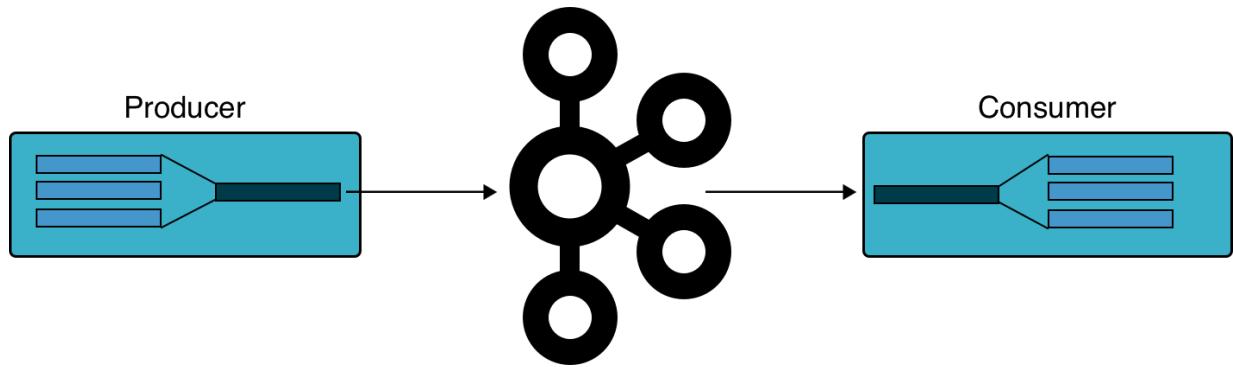
# Batching Messages (2)

- `batch.size` (Default: 16KB):
    - The maximum size of a batch before sending
  - `linger.ms` (Default: 0, i.e., send immediately):
    - Time to wait for messages to batch together
- 

When a message is pushed from the queue to the Brokers is determined by either how long the messages have been in the queue (`linger.ms`) or the amount of data (`batch.size`). The default behavior is for the Producer to push messages in real-time so `linger.ms` defaults to 0; i.e., send message as soon as they arrive.

tejaswin.renugunta@walgreens.com

# End-To-End Batch Compression



1. Producer batches and compresses into single message
2. Compressed message stored in Kafka
3. Consumer decompresses

The compression type used by a Producer is noted as an attribute in the messages that it produces. This allows multiple Producers writing to the same Topic to use different compression types. Consumers will decompress messages according to the compression type denoted in the header of each message.

# Tuning Producer Throughput

- `batch.size, linger.ms`
  - High throughput: large `batch.size` and `linger.ms`, or flush manually
  - Low latency: small `batch.size` and `linger.ms`
- `buffer.memory`
  - Default: 32MB
  - The Producer's buffer for messages to be sent to the cluster
  - Increase if Producers are sending faster than Brokers are acknowledging, to prevent blocking
- `compression.type`
  - `gzip, snappy, lz4, zstd`
  - Configurable per Producer, Topic, or Broker

---

A typical setting for batching is `linger.ms=100` and `batch.size=1000000`.

The `buffer.memory` should be larger than your target batch size accumulated across all target Partitions. For example, if the Topic has 10 Partitions and an expected 16KB batch size, the minimum size for the buffer is 160KB. This should typically be set larger to allow for buffering in the event of retries when pushing data to the Brokers.

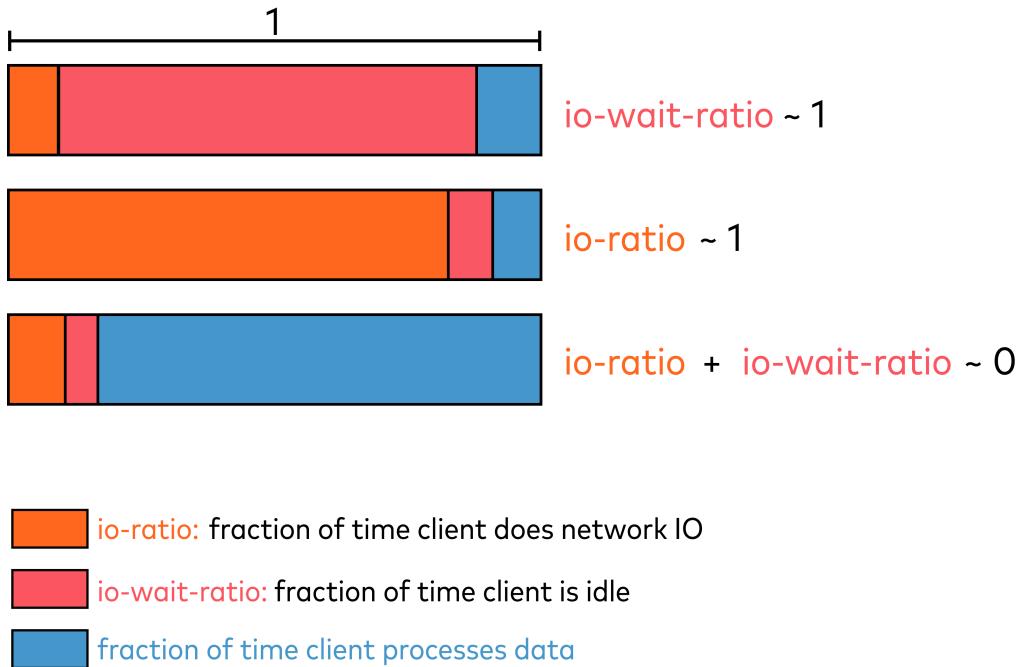
The `compression.type` can be set on Brokers, Topics, or Producers. On Brokers and Topics, the default is `compression.type=producer`, which means the compression codec of the Producer is respected. This can result in a single Topic containing messages of various compression types. If the Broker or Topic has its own compression type set, then all messages will be compressed with the specified codec. This puts extra work on the Broker, but will save space. Disk space is usually relatively cheap compared to compute time, so usually it is best to leave Broker and Topic `compression.type` settings to `producer`.

The Zstandard compression type was added in Kafka 2.1.



Batch size refers to the compressed size if compression is enabled.

# Monitoring Client Performance



Sometimes, a Producer/Consumer client may appear slow. The slowness could be caused by either the client or the Broker. Before tuning the system, it is recommended to first identify where the bottleneck is. There are 2 JMX metrics in Kafka Producer/Consumer:

- **io-ratio**: The fraction of the time that the client is spent on producing/retrieving the data to/from the Broker.
- **io-wait-ratio**: The fraction of the time that the client is idle.

Both values are between 0 and 1, and the sum of the two values is no more than 1. The rest of the time is spent by the client application.

Here are some guidelines to help analyze **io-ratio** and **io-wait-ratio**:

- If **io-wait-ratio** is close to 1, it indicates that the client is mostly idle and the bottleneck is likely on the Broker.
- If **io-ratio** is close to 1, it indicates that the client is mostly busy interacting with the Brokers. If the client is a Producer, it may be appropriate to do more batching or compression to increase throughput. If the client is a Consumer, it may be appropriate to increase `max.partition.fetch.bytes` or increase the size of the Consumer Group to achieve higher throughput. Remember that the most Consumers that one can run in a Consumer Group is limited by the number of Partitions in the consumed Topics.
- If **io-ratio** and **io-wait-ratio** are both close to 0, it indicates that the client is the bottleneck. For Producers, make sure that the Producer callback is not doing expensive

operations (e.g., writing to a log4j file). For Consumers, make sure that there is no expensive step in processing each returned record.

tejaswin.renugunta@walgreens.com

# Client Performance Metrics

- Client-level JMX metrics:

```
kafka.producer:type=producer-metrics,client-id=my_producer  
kafka.consumer:type=consumer-metrics,client-id=my_consumer
```

- Producer-only metrics:

`batch-size-avg`

`compression-rate-avg`

- Per-Topic metrics:

```
kafka.producer:type=producer-topic-metrics,client-id=my_producer,topic=my_topic
```

`record-send-rate`

`byte-rate`

`record-error-rate`

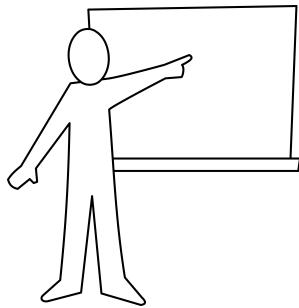
The `batch-size-avg` and `compression-rate-avg` metrics can verify the effectiveness of your tuning:

- A `batch-size-avg` much smaller than `batch.size` indicates inefficient batching, so consider increasing `linger.ms`.
- A low `compression-rate-avg` would indicate that compression is not creating much space savings and so may not be worth the system resources to run the compression.

Terminology note: Though most of the documentation refers to the key-value pairs as "messages" or "events", the APIs and metrics frequently refer to them as "records".

For an exhaustive list of both Broker and client metrics, see  
<https://kafka.apache.org/documentation/#monitoring>

# Module Map



- Terminology Review
- Producer Performance
-  Hands-on Lab: **Exploring Producer Performance**
- Broker Performance ... 
- Broker Failures and Recovery Time
- Load Balancing Consumption
-  Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance
- Performance Testing
-  Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# Monitoring Leaders and Partitions

- Monitoring with JMX Metrics

- **LeaderCount** (gauge)

```
kafka.server:type=ReplicaManager,name=LeaderCount
```

- **PartitionCount** (gauge)

```
kafka.server:type=ReplicaManager,name=PartitionCount
```



Leadership and Partitions should be spread evenly across Brokers

---

It's worth mentioning that a "gauge" is a measure of a value taken at a specific point in time, whereas a "meter" is a measure of rate.

**LeaderCount** and **PartitionCount** are critical to monitor performance. The number of leaders should be relatively similar across all Brokers.

**PartitionCount** includes all replicas, regardless of role (leader or follower).

**LeaderCount** is not an absolute measure of balanced performance. A heavily-used Leader and a Leader with no client requests carry the same weight in this metric.

# Network Threading on the Broker

- Brokers process requests as follows:

Operation	Request Type	Metric Name
Producers write data to the cluster	ProduceRequest	Produce
Consumers read data from the cluster	FetchRequest	FetchConsumer
Replicas read data from their Leaders	FetchRequest	FetchFollower

- JMX metrics:

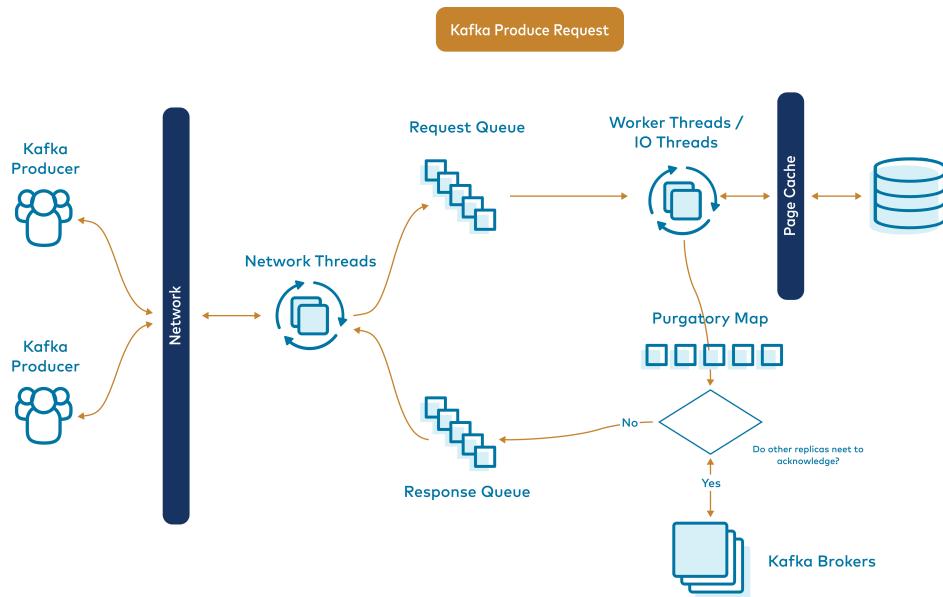
```
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Produce  
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=FetchConsumer  
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=FetchFollower
```

There are three types of network threads on the Brokers:

- **ProduceRequests**: write requests from Producers
- **FetchRequest**s: read requests from Consumers
- **ReplicaFetchRequests**: replication requests from follower Brokers

Brokers process these requests in parallel using threads.

# Anatomy of a Produce Request on a Broker



This diagram shows the flow of a `ProduceRequest` through a Broker.

Purgatory (described on the next slide) is a waiting area for requests that are waiting for other replicas to confirm. The only time this will be used is if the Producer was configured with `acks=all` and the Broker has to wait until the data is committed (written to all members of the ISR list) before sending a response back to the Producer.

# Purgatory

- Produce purgatory:
  - Produce requests waiting for acks from other replicas
- Fetch purgatory:
  - Fetch requests waiting for more bytes



Completed requests need to be garbage-collected

Purgatory is a memory-resident structure used for stashing requests that cannot be completed immediately. There are separate purgatories for `ProduceRequest` and `FetchRequest` data.

Purgatory typically runs well with the default configurations. However, consider tuning purgatory to have more aggressive garbage collection if the size of the map continuously grows.

# Message Size Limit



Avoid changing maximum message size. Kafka is not optimized for very large messages.

## Brokers or Topics

- `message.max.bytes` - maximum size of message that Brokers can receive from a Producer (Default: 1MB)

## Replication on Brokers

- `replica.fetch.max.bytes` - maximum amount of data per-Partition that Brokers send for replication (Default: 1MB)

## Consumers

- `max.partition.fetch.bytes` - maximum amount of data per-Partition the Broker will return (Default: 1MB)

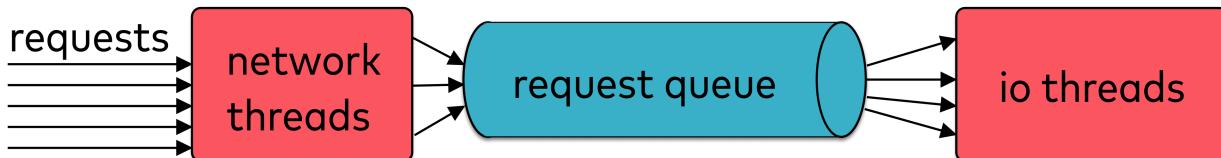
The configurations shown are the settings related to message size. Kafka is optimized for the default 1MB maximum message size. If a Broker receives a large message, a byte buffer must be allocated to receive the entire message, which could cause problems such as fragmentation in the heap. If a larger message size is required, consider alternatives such as compression, breaking the message into smaller pieces, or sending a reference to the object (e.g., a storage location for a file).

If the maximum message size was changed in Kafka prior to 0.10.1.0, the setting `max.partition.fetch.bytes` (maximum bytes returned per Broker, per Partition) had to be changed as well. That setting was a hard limit to control the memory use in the Consumers. A similar setting (`replica.fetch.max.bytes`) had to be adjusted to allow replication of the larger messages as well.

As of Kafka 0.10.1.0, both settings are now non-absolute (soft) limits. If the first message in the Partition of the fetch is larger than this limit, the message will still be returned to ensure that the Consumer or replica can make progress.

# Performance Tuning the Thread Pools

- Each thread pool is configurable
  - `num.network.threads` (increase for SSL)
  - `num.io.threads`



- If there are no available I/O threads, requests can be queued up to `queued.max.requests`
  - Default is 500
  - Match with the number of clients per Broker
- If the Request Queue is filled, the network threads stop reading in new requests

Sizing the Request Queue to number of clients connecting to the Broker gives every client the chance to buffer a request. By default queue size is about 500, which is plenty for most use cases.

If the request queue is full, it can affect the response processing since the inbound and outbound network connections share the network thread pool. If an incoming `ProducerRequest` (write) is blocked because the queue is full, the request will occupy one of the network threads as it retries. If the network threads are all occupied, the Broker will not be able to take additional incoming requests, but also cannot process outgoing responses even if they are ready. A small request queue will affect all aspects of the Broker's network connections.

The `num.network.threads` setting is per port. The default value is 3. This configuration must be increased for SSL because of the increased CPU cost due to encryption/decryption over the wire.

The `num.io.threads` default value is 8.

# Monitoring Thread Capacity

- JMX metrics:

```
kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent (meter)  
kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent (gauge)
```

- 0 indicates all resources are used
  - 1 indicates all resources are available
- Alert if the value drops below 0.4
    - If it does, consider increasing the number of threads

---

I/O threads and network threads impact parallelism and performance. Knowing how much capacity is left in the thread pools is important to prevent running out of resources.

Brokers maintain metrics which report the percentage of idle capacity in the network thread pool ([NetworkProcessorAvgIdlePercent](#)) and I/O thread pool ([RequestHandlerAvgIdlePercent](#)). The value for these metrics range between 0 and 1, where 1 means all resources are available/idle and 0 means all resources are used. Set your monitoring tools to alert if these values drop below 0.4 or 0.3 (40% or 30% idle; 60% or 70% used). Dropping below these levels indicate heavy utilization and may require additional capacity - either increase the size of the pool or grow the cluster.



The metrics occasionally show values >1.0 due to rounding errors.

# Monitoring The Request Queue

- JMX metrics:

```
kafka.network:type=RequestChannel,name=RequestQueueSize  
kafka.network:type=RequestMetrics,name=RequestQueueTimeMs
```

- Congested request queue can't process incoming or outgoing requests

---

As discussed earlier, a full request queue will affect all types of network requests through the Broker, including both produce and fetch. The default setting of 500 for `queued.max.requests` should be good enough for most use cases. If the listed metrics show the maximum request queue size is being reached or that requests are spending too long in queue, then it may be necessary to consider:

- Finding ways to reduce the frequency of page cache flushing to disk
  - e.g. increasing `log.segment.bytes` on the Brokers
- Increasing `queued.max.requests` at the cost of added memory pressure
- Increasing `num.io.threads`
- Investigating other possible IO issues
- Scaling the cluster horizontally with more Brokers
- Scaling the cluster vertically with faster IO disks

# Metrics for Monitoring Requests

- Monitor the total time for produce and fetch requests

```
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Produce
```

```
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=FetchConsumer
```

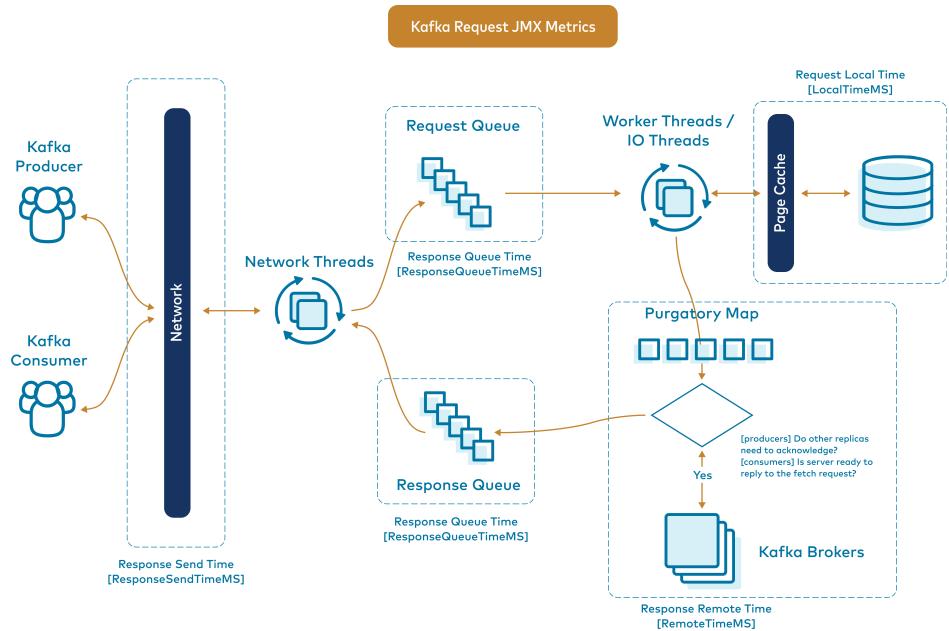
```
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=FetchFollower
```

The first metric to check when troubleshooting latency issues is to look at the time it takes for a request to travel through the Broker. The **TotalTimeMs** metric exposes this information and can show differences between the different types of requester: Producer, Consumer, Replica (Follower).



It is important to have benchmarks for your environment to know whether the observed readings are within expected performance for your specific environment. Benchmarking performance is discussed later in this chapter.

# Monitoring Requests on the Broker



Because the **ProduceRequest** is handled by multiple components within the Broker, a slow down at any of them will increase the overall latency for the request.

# Request Lifecycle and Latencies

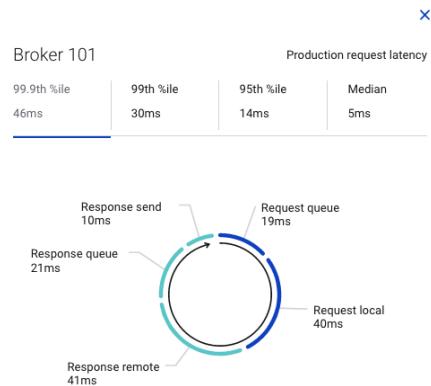
Break down `TotalTimeMs` further to see the entire request lifecycle

Metric	Description
<code>RequestQueueTimeMs</code>	Time the request waits in the request queue
<code>ResponseSendTimeMs</code>	Time to send the response
<code>ResponseQueueTimeMs</code>	Time the request waits in the response queue
<code>LocalTimeMs</code>	Time the request is processed at the leader
<code>RemoteTimeMs</code>	Time the request waits for the follower

tejaswin.renugunta@walgreens.com

# Request Lifecycle and Latencies

Confluent Control Center:



Each metric is a percentile, and percentile metrics don't add associatively.

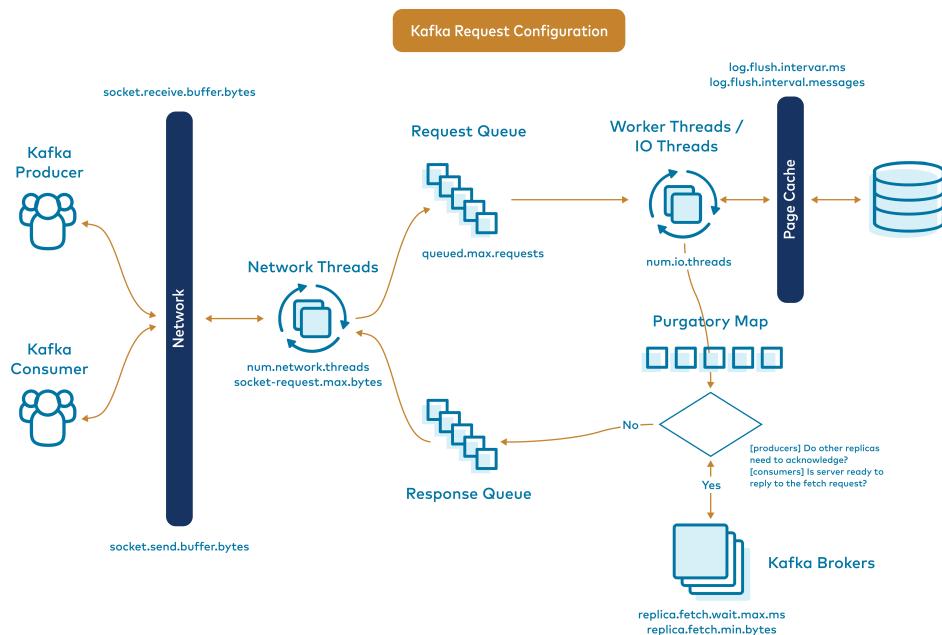
Once you have determined that the total time for the request is too large, isolate the bottleneck by viewing the metrics for each of the components. Since downstream backups can affect components earlier in the sequence, consider checking the metrics in the order:

- `ResponseSendTimeMs`
- `ResponseQueueTimeMs`
- `RemoteTimeMs`
- `LocalTimeMs`
- `RequestQueueTimeMs`

These metrics can display the values as averages and some percentiles (50th, 95th, 99th, 99.9th).

In the Producer case, `kafka.network:type=RequestMetrics, name=ResponseSendTimeMs` time should be pretty small.

# Configuring Requests on the Broker

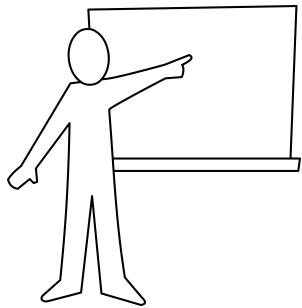


Once you determine where the bottleneck is, consider making changes to the Broker to fix the issue. This slide lists some of the properties that can be changed for each of the components in the request flow.



Kafka level log flushing is disabled by default because modern Linux operating systems are more optimized for page cache flushing. If IO is a bottleneck, see again the previous slide about request queue metrics.

# Module Map



- Terminology Review
- Producer Performance
-  Hands-on Lab: **Exploring Producer Performance**
- Broker Performance
- Broker Failures and Recovery Time ... 
- Load Balancing Consumption
-  Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance
- Performance Testing
-  Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# Avoiding Soft Failures from Garbage Collection



- Garbage Collection can cause:
  - unnecessary Consumer Group rebalances
  - unnecessary Partition leader elections on Brokers
- Enable GC logging in the Broker JVM
  - set environment variable `GC_LOG_ENABLED="true"` and restart Broker

- Monitor ZooKeeper timeouts with JMX:

```
kafka.server:type=SessionExpireListener,name=ZooKeeperExpiresPerSec
```

Garbage collection is a background Java thread that pauses processes to reclaim unused memory. Long Garbage Collection periods in the Broker should be avoided. If GC exceeds the Broker property `zookeeper.session.timeout.ms` (18 seconds, by default), the Broker will appear to be offline and the leaders hosted by that Broker will be re-elected unnecessarily. Also, if a Consumer has a session timeout with the Consumer Group Coordinator (a Broker), it will leave and rejoin its Consumer Group. The Broker setting `zookeeper.session.timeout.ms` determines how long a Broker can be offline before it is considered dead (default 18 seconds). Monitor `ZooKeeperExpiresPerSec` for any value greater than 0.

Recommendations:

- Use the G1 Garbage Collector (available since Java 7). Newer Java 11 garbage collectors like the Z Garbage Collector have not been tested as of Kafka 2.2, but may be worth exploring.
- Enable GC Logging. If Brokers are losing sessions from ZooKeeper (shown as session expiration errors in the `server.log`), there is likely a ZooKeeper connection problem. Enabling GC logging will help determine if long GC times are the cause. Long GC times are one of the most common reasons for session expiration.

- Parse and analyze the GC logs here: [http://gceeasy.io/](http://gceasy.io/)
- Check that the connection between Brokers and ZooKeeper is good. Otherwise, ZooKeeper may falsely detect a Broker as dead.

For more information about tuning garbage collection, see  
<https://docs.confluent.io/current/kafka/deployment.html#jvm>.

tejaswin.renugunta@walgreens.com

# Recovering from a Broker Failure

Cluster lives on:

- New Partition leaders elected
  - `time ~ #P/#B`
- Sometimes new Controller elected
  - `time ~ #P`

Broker gets back in the game:

- Administrator replaces failed hardware
- Broker automatically recovers Partition data starting from checkpoint



Broker recovery is CPU, I/O, and bandwidth intensive, especially if there are many Partitions per Broker. If possible, recover a Broker or perform data rebalances during off-peak times.

Kafka is highly resilient to failures. If a Broker fails, any Partition for which the Broker was the leader must find a new leader through leader election. There is some temporary degradation in performance while the Controller facilitates leader elections, but otherwise the cluster will continue to function. Time for new leader election is proportional to the number of Partitions per Broker. In the case that the failed Broker was the Controller, a new Controller is elected in time proportional to the total number of Partitions in the cluster.

If a Broker had a hard failure, the first thing it needs to do is log recovery. The Broker doesn't know when it failed or what state the log is in. For each Partition, the Broker will read the last flushed offset from the `recovery-point-offset-checkpoint` file and read every message from that offset to the end of the Commit Log to verify the CRC. At the first offset where the CRC doesn't match, the Broker will truncate the log from that point on and start replicating from the Leader.

Because of the amount of reads and writes, log recovery can be I/O intensive. Recovery may also be CPU intensive if the logs are compressed. Recovery time will be improved by using multiple threads, one per Partition. By default, the Broker starts one recovery thread per Partition directory, which is sufficient for deployments where there is one disk per log directory. For RAID environments, change `num.recovery.threads.per.data.dir` to match the number of disks you have in the RAID set or number of CPU cores. For cloud-based hosting environments where the number of disks used for storage is unknown, match the setting to the number of CPU cores. Matching the setting to number of CPU cores is also a useful recommendation when using SSDs for storage.

# Discussion

- You notice that Produce requests keep stalling. This happens regularly. Stalls are for 6-10 seconds. Your users have noticed that there are several duplicate messages after the stalls. Acks are set to `all`. What do you think might be going on, and how would you investigate further?

---

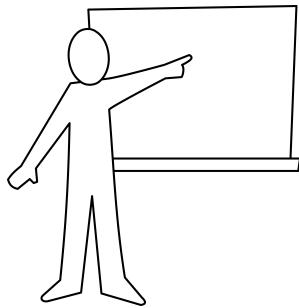
Take several minutes to consider the question. Consider consulting configuration and metrics documentation:

- <https://docs.confluent.io/current/installation/configuration/index.html#configuration-reference>
- <https://docs.confluent.io/current/kafka/monitoring.html>

Here are some ideas related to the discussion:

- ISR fluctuations might explain the pauses, depending on the value of `replica.lag.time.max.ms`, but the presence of duplicate messages implies leaders are being re-elected.
- To investigate:
  - Check the ControllerStats `LeaderElectionRateAndTimeMs` metric to see if the Controller is moving leadership. If so, it could be due to leaders becoming unresponsive. Check the Broker's `server.log` for entries like `session expired` and ZooKeeper registration.
  - Check the replica `MaxLag` metric to see if it goes beyond `replica.lag.time.max.ms`. If so, then followers will be ejected from the ISR
  - Check RequestMetrics `RemoteTimeMs` metric to see how long messages take to replicate to followers. If there is a spike here, there is something going wrong with replication.
    - Correlate with settings for `replica.lag.time.max.ms`.
  - Enable garbage collection logging and check Java garbage collection times to see if they are greater than the ZooKeeper timeout.
  - If the problem appears to be unique to a single Broker, check `RequestQueueSize` to see if the request queue on the Broker is congested.

# Module Map

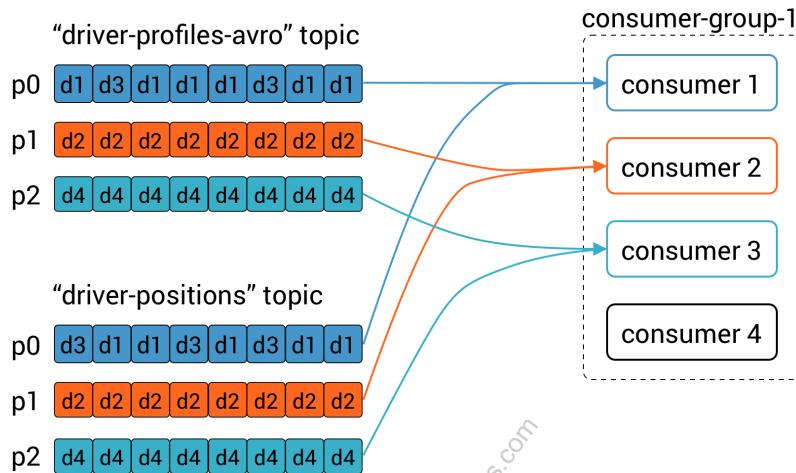


- Terminology Review
- Producer Performance
-  Hands-on Lab: **Exploring Producer Performance**
- Broker Performance
- Broker Failures and Recovery Time
- Load Balancing Consumption ... 
-  Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance
- Performance Testing
-  Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# Load Balancing Across Consumers

- Consumers are identified to a Consumer Group according to `group.id`, a property defined in Consumer code
  - Consumers in the group are typically on separate machines
  - Subscribed Topic partitions are divided among group members



Kafka Topics allow the same message to be consumed multiple times by different Consumer Groups.

A Consumer Group binds together multiple Consumers for parallelism.

- Members of a Consumer Group are subscribed to the same Topic(s).
- The Partitions will be divided among the members of the Consumer Group to allow the partitions to be consumed in parallel. This is useful when processing of the consumed data is CPU intensive or the individual Consumers are network-bound.
- Within a Consumer Group, a Consumer can consume from multiple Partitions.
- A Partition is only assigned to one Consumer to prevent repeated processing of record data.

Consumer Groups have built-in logic which triggers partition assignment on certain events, e.g., Consumers joining or leaving the group.

# Partition Assignment within a Consumer Group

- Partitions are 'assigned' to Consumers
- A single Partition is consumed by only one Consumer in any given Consumer Group
  - messages with same key will go to same Consumer (unless you change number of Partitions)
  - `partition.assignment.strategy` in the Consumer configuration

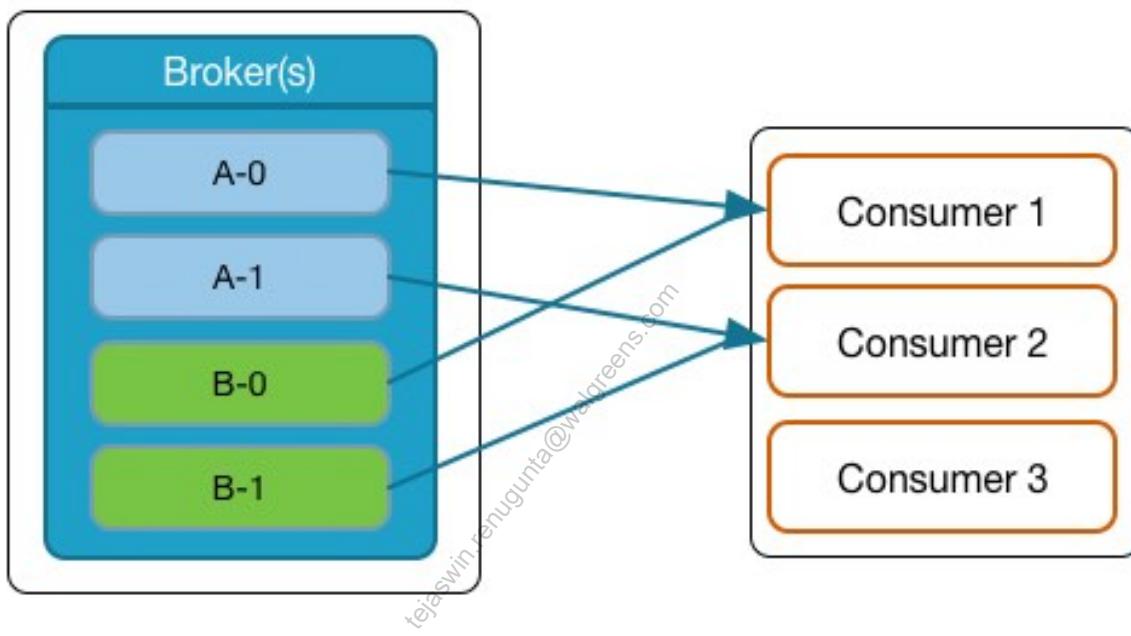
---

The Consumer Group is managed by a process called a Group Coordinator, which will be discussed in detail later in the course.

tejaswin.renugunta@walgreens.com

# Partition Assignment Strategy: Range

- Range is the default `partition.assignment.strategy`
- Useful for co-partitioning across Topics:
  - Package ID across `delivery_status` and `package_location`
  - User ID across `search_results` and `search_clicks`



There are four built-in Partition Assignment strategies: Range (default), RoundRobin, Sticky and CooperativeSticky. These are set with `partition.assignment.strategy` in the Consumer code.

The Range strategy is useful for "co-partitioning" across Topics with keyed messages. Imagine that these two Topics are using the same key—for example, a `userid`. Topic A is tracking search results for specific userids; Topic B is tracking search clicks for the same set of userids. If the Topics have the same number of Partitions and messages are partitioned with the default `hash(key) % number of Partitions`, then messages with the same key would land in the same numbered Partition in each Topic. Therefore the Range strategy will ensure messages with a given `userid` from both Topics will be read by the same Consumer.

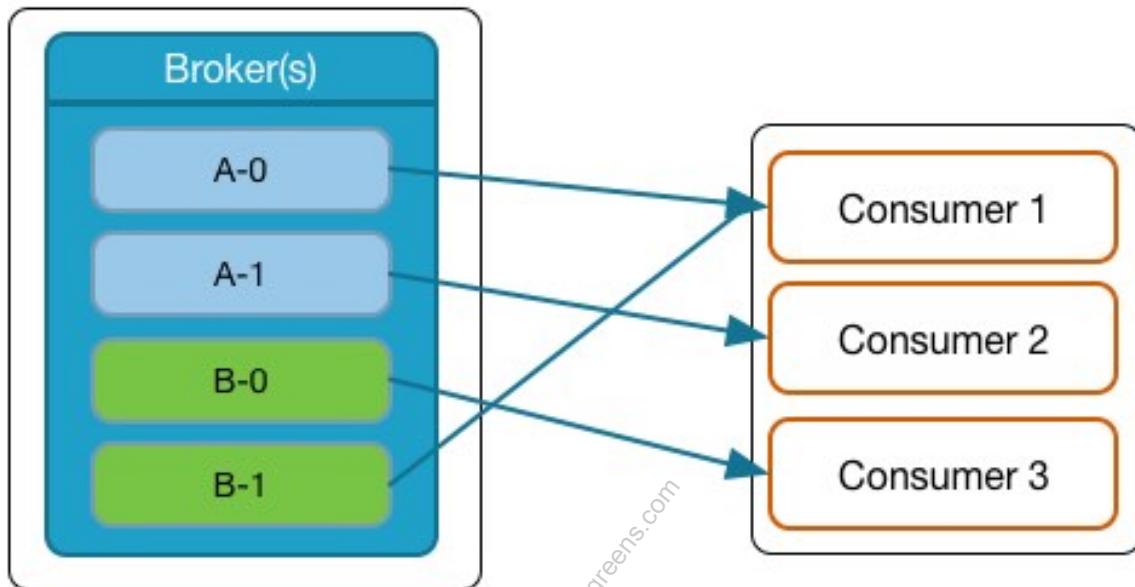
The Range strategy assigns matching Partitions of different Topics to the same Consumer. In the image shown, there are two 2-Partition Topics and three Consumers. The first Partition from each Topic is assigned to one Consumer, the second Partition from each is assigned to another Consumer, repeating until there are no more Partitions to assign. Since

we have more Consumers than Partitions in any Topic, one Consumer will be idle.

tejaswin.renugunta@walgreens.com

# Partition Assignment Strategy: RoundRobin

- Partitions assigned one at a time in rotating fashion



The RoundRobin `partition.assignment.strategy` is much simpler. Partitions are assigned one at a time to Consumers in a rotating fashion until all the Partitions have been assigned. This provides much more balanced loading of Consumers than Range.

An important note about Range and RoundRobin: Neither strategy guarantees that Consumers will retain the same Partitions after a reassignment. In other words, if a Consumer 1 is assigned to Partition A-0 right now, Partition A-0 may be assigned to another Consumer if a reassignment were to happen. Most Consumer applications are not locality-dependent enough to require that Consumer-Partition assignments be static.

# Partition Assignment Strategy: Sticky and CooperativeSticky

- Sticky
    - Is RoundRobin with assignment preservation across rebalances
  - CooperativeSticky
    - Is Sticky without its "stop-the-world" rebalancing of all partitions
- 

- Sticky

If your application requires Partition assignments to be preserved across rebalances, use the Sticky strategy. Sticky is RoundRobin with assignment preservation. The Sticky strategy preserves existing Partition assignments to Consumers during rebalances to reduce overhead:

- Kafka Consumers retain pre-fetched messages for Partitions assigned to them before a rebalance
- Reduces the need to clean up local Partition state between rebalances

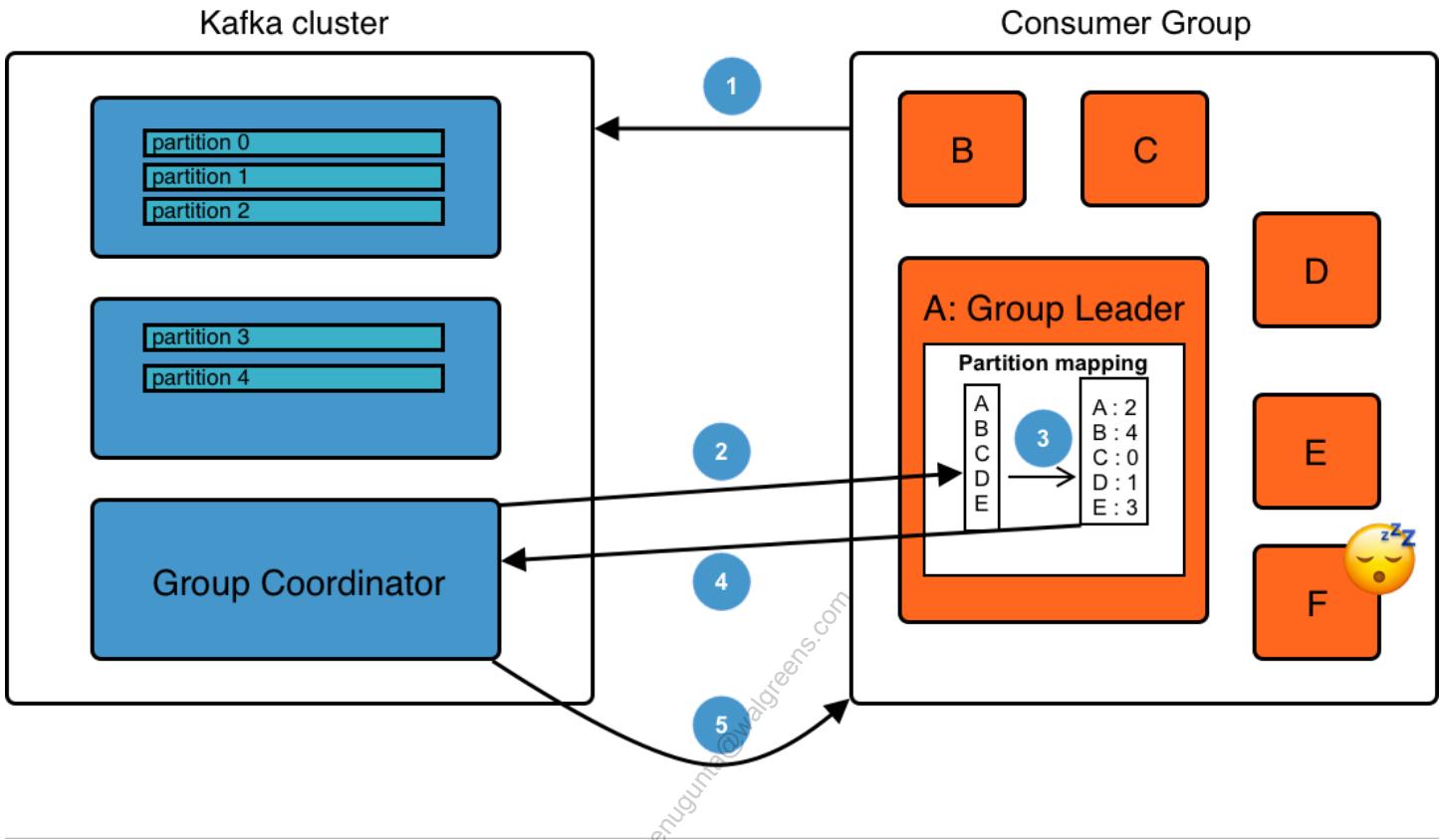
See <https://cwiki.apache.org/confluence/display/KAFKA/KIP-54+-+Sticky+Partition+Assignment+Strategy> [KIP 54] for a full description of this feature.

- CooperativeSticky

At a high level, it is very similar to Sticky but it uses consecutive rebalances rather than the single stop-the-world used by Sticky.

See <https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%3A+Kafka+Consumer+Incremental+Rebalance+Protocol> [KIP 429] for a full description of this feature.

# Registering a Consumer Group



There are two major components that facilitate Consumer Group coordination: The Group Coordinator (a Broker), and the Group Leader (a Consumer). The process of registering a Consumer Group and assigning Partitions across Consumers is as follows:

1. Each Consumer registers itself with the Kafka cluster using its `group.id`. Kafka creates the Consumer Group and all offsets of these Consumers will now be stored on a Partition of the special `__consumer_offsets` Topic. The lead Broker for this Partition is selected to be the Group Coordinator for the Consumer Group.
2. The Group Coordinator waits `group.initial.rebalance.delay.ms` (default 3 seconds) for `JoinGroup` requests from Consumers before creating a catalog of Consumers in the Group. In the image, this is represented by the list "A,B,C,D,E". The Group Coordinator will list Consumers in the order it receives `JoinGroup` requests up to a maximum of the number of Partitions the Consumer Group will be consuming from.



Consumer F is not listed because there are only 5 Partitions and F wasn't one of the first 5 Consumers to join the Group. Consumer F will be idle.

3. The first Consumer to send a `JoinGroup` request will be named the Group Leader. In this

case, it is Consumer A. The Group Leader receives the list of Consumers from the Group Coordinator. Then, the Group Leader uses its configured `partition.assignment.strategy` to assign Partitions to each Consumer. Here, there are more Consumers than Partitions, so each Consumer will have at most 1 Partition to consume from. If there were more Partitions than Consumers, then Consumers would be assigned multiple Partitions.



One might ask, "why doesn't the Group Coordinator assign Partitions itself?" One principle of Kafka is to offload as much computation as possible to clients. With many Consumer Groups and rebalances, it is best to offload these computations to the client.

4. The Group Leader sends the mapping Consumers → Partitions back to the Group Coordinator. The Group Coordinator caches the mapping in memory and persists it in ZooKeeper.
5. The Group Coordinator sends each Consumer the Partitions it has been assigned.

Kafka keeps track of Consumer offsets in a special Topic called `__consumer_offsets` which is partitioned by the `group.id` of Consumer Groups. When a Consumer Group is registered, the leader for the Partition of `__consumer_offsets` where the `group.id` lands becomes the Group Coordinator. If the Group Coordinator fails, there is a leader election of this `__consumer_offsets` Partition and the new leader becomes the new Group Coordinator. This is all handled by Kafka automatically. One can determine which Partition a `group.id` lands on with the formula `hash(group.id) % offsets.topic.num.partitions`, where `offsets.topic.num.partitions` is the cluster-wide Broker configuration property. From there, one can determine the Group Coordinator by finding the leader for the computed Partition.

You can check which Broker is the Group Coordinator for a given `group.id` with:

```
$ kafka-consumer-groups \
--bootstrap-server <broker>:<port> \
--describe --group <group.id> \
--state
```

# Consumer Rebalancing

## Rebalance triggers:

- Consumer leaves Consumer group
- New Consumer joins Consumer Group
- Consumer changes its Topic subscription
- Consumer Group notices change to Topic metadata (e.g. increase Partitions)

## Rebalance Process:

1. Group Coordinator uses flag in heartbeat to signal rebalance to Consumers
2. Consumers pause, commit offsets
3. Consumers rejoin into new "generation" of Consumer Group
4. Partitions are reassigned
5. Consumers resume from new Partitions



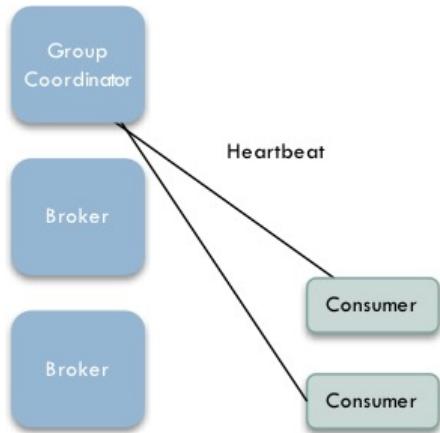
Consumption pauses during rebalance. Avoid unnecessary rebalances.

During rebalance, Consumers will stop reading from their current Partitions, commit their Consumer offsets, receive their new assignment, then resume reading from their newly assigned Partitions.

Once a rebalance has begun, the coordinator starts a timer which is set to expire after the group's session timeout. Each member in the previous generation detects that it needs to rejoin with its periodic heartbeats sent to the coordinator. The coordinator uses the **REBALANCE\_IN\_PROGRESS** error code in the heartbeat response so the Consumer knows to rejoin.

Partitions are reassigned in the same way they were assigned in the first place: with a dance between Group Coordinator and Group Leader. There may be a new Group Leader on rebalance.

# Consumer Failure Detection



- Consumers send heartbeats in background thread, separate from `poll()`
  - `heartbeat.interval.ms` (Default: 3 sec)
- `session.timeout.ms` (Default: 10 sec)
  - If no heartbeat is received in this time, Consumer is dropped from Consumer Group
- `poll()` must still be called periodically
  - `max.poll.interval.ms` (Default: 5 minutes)

The Consumer communicates to the Brokers regularly to let them know that it is still alive. If a Consumer is believed to be dead, it is removed from the Consumer Group and a Consumer rebalance is performed.

# Avoiding Excessive Rebalances

- Tune `session.timeout.ms`:
  - set `heartbeat.interval.ms` to 1/3 `session.timeout.ms`
  - Pro: gives more time for Consumer to rejoin
  - Con: takes longer to detect hard failures
- Tune `max.poll.interval.ms`
  - Give Consumers enough time to process data from `poll()`

Static group membership:

- Assign each Consumer in Group unique `group.instance.id`
- Consumers do not send `LeaveGroupRequest`
- Rejoin doesn't trigger rebalance for known `group.instance.id`



These configurations are set in Consumer code.

Consumer Group rebalances can be costly because each rebalance pauses consumption for all Consumers in the Group. Furthermore, rebalances can also involve shuffling the Partitions assigned to each Consumer, which is unacceptable in more stateful applications where sticky Partition assignment matters. The Sticky assignment strategy can only be used with round robin assignment and only reduces assignment changes rather than avoiding rebalance.

In order to avoid unnecessary rebalances, you can tune `session.timeout.ms` and `heartbeat.interval.ms` (with `heartbeat.interval.ms` recommended to be 1/3 of `session.timeout.ms`). You might also consider enabling static group membership by configuring a unique `group.instance.id` to each Consumer in the Group.

Increasing `session.timeout.ms` means it will take longer for the Group Coordinator to declare a Consumer dead and remove it from the Group, thus postponing the rebalance. This gives time for the Consumer to recover and pick up where it left off. The downside to this is that it will now take longer for the Group Coordinator to detect hard Consumer failures that actually do require rebalance. The Partitions assigned to the failed Consumer will go unconsumed for this time.



The `session.timeout.ms` property must be between the `group.min.session.timeout.ms` (Default: 6 seconds) and `group.max.session.timeout.ms` (Default: 5 minutes) properties on the Broker.

The Consumer must also `poll()` periodically to remain in the Consumer Group, so set `max.poll.interval.ms` high enough to allow the Consumer to process the data it receives from `poll()`.



Do not just turn `max.poll.interval.ms` up to a very high value, since in a case where your main thread crashed but the background heartbeat continued, this would result in waiting for that long amount of time before a Consumer rebalance could be triggered.

Static group membership can also help to avoid unnecessary rebalances during rolling updates or restarts (e.g. ephemeral containers in Kubernetes or the rollout of a new application version). This normally would involve two rebalances: one rebalance when the Consumer leaves the Group, and another rebalance when it rejoins the Group. With static group membership, a Consumer does not send a `LeaveGroupRequest` to the Group Coordinator, thus avoiding the first rebalance. Then, when the Consumer rejoins the Group, the Group Coordinator sees its `group.instance.id` and does not trigger a rebalance. The Consumer simply picks up where it left off.



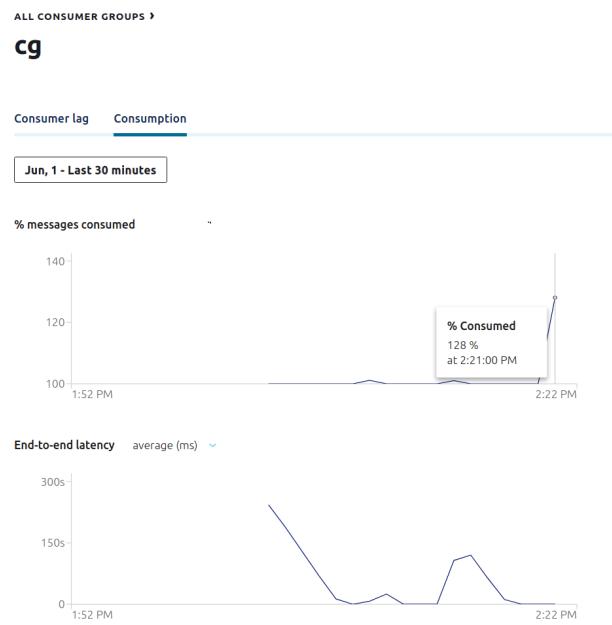
Assigning a unique `group.instance.id` per Consumer can itself be laborious without automation. One strategy in Kubernetes would be to deploy the Consumer Group as a StatefulSet and assign `group.instance.id` as the pod name of the Consumer. This can be accomplished by exposing the pod name to the Consumer container as an environment variable and setting `group.instance.id` equal to that environment variable. For example, see <https://kubernetes.io/docs/tasks/inject-data-application/environment-variable-expose-pod-information/>.

Most features of static group membership were released in Apache Kafka 2.3. Kafka Connect uses a version of it for cooperative task rebalancing. See:

- <https://cwiki.apache.org/confluence/display/KAFKA/KIP-345%3A+Introduce+static+membership+protocol+to+reduce+consumer+rebalances>
- And the first 5 minutes of [https://www.youtube.com/watch?v=57Jf\\_9IrlwA&list=PLa7VYiOyPIHOsnucuYWkuUXwasMr-HR7Y&index=4&t=0s](https://www.youtube.com/watch?v=57Jf_9IrlwA&list=PLa7VYiOyPIHOsnucuYWkuUXwasMr-HR7Y&index=4&t=0s).

# Under Consumption and Over Consumption

- Under-consumption:
  - Consumer offsets intentionally set to latest, skipping old messages
  - Misbehaving Consumers, e.g. did not follow shutdown sequence
- Over-consumption:
  - Consumers reprocessing data, e.g. machine learning model testing
- Monitor under/over-consumption in Confluent Control Center



Confluent Control Center enables administrators to visually track over and under consumption. This helps to troubleshoot issues if Consumers are reporting missing or repeated messages.

# When a Consumer Group is Empty (1)

- When all Consumers leave a Group, the Group metadata is deleted from Coordinator
- Verify with:

```
$ kafka-consumer-groups --bootstrap-server=broker-101:9092  
--list
```

---

The Consumer Group metadata (e.g. offset information) for a given Consumer Group is maintained by the Group Coordinator. We can discover the current Group Coordinator for a group by issuing a "Group Coordinator Request". If the Consumer Group still has Consumers, then there will be a Group Coordinator to respond with metadata to the `kafka-consumer-groups` call. If the Consumer Group does not have Consumers, then the `kafka-consumer-groups` call will not return anything.

tejaswin.renugunta@walgreens.com

# When a Consumer Group is Empty (2)

- `__consumer_offsets` Topic has its own retention policy:
  - cleanup policy: `compact`, but
  - `offsets.retention.minutes` (Default: 10080 minutes [=1 week]) applies after Group is empty
- View the `__consumer_offsets` Topic:

```
$ kafka-console-consumer \
>   --topic __consumer_offsets \
>   --bootstrap-server kafka-1:9092,kafka-2:9092 \
>   --formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter"
...
[new-group,new-topic,0]::OffsetAndMetadata(offset=3, leaderEpoch=Optional[0],
metadata=, commitTimestamp=1565702696548, expireTimestamp=None)
..."
```

After a Consumer Group loses all its Consumers (i.e. becomes empty) its offsets will be kept for the `offsets.retention.minutes` period before getting discarded. For standalone Consumers (using manual assignment), offsets will be expired after the time of last commit plus this retention period.



The default retention period for Consumer offsets was changed in AK 2.0 from 1440 minutes (1 day) to 10080 minutes (1 week)

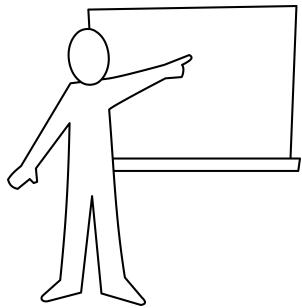
# Hands-On Lab

- In this Hands-On Exercise, you will increase the number of Partitions in a Topic and view offsets in an active Consumer Group.
- Please refer to **Lab 05 Optimizing Kafka's Performance** in the Exercise Book:
  - a. **Modifying Partitions and Viewing Offsets**



tejaswin.renugunta@walgreens.com

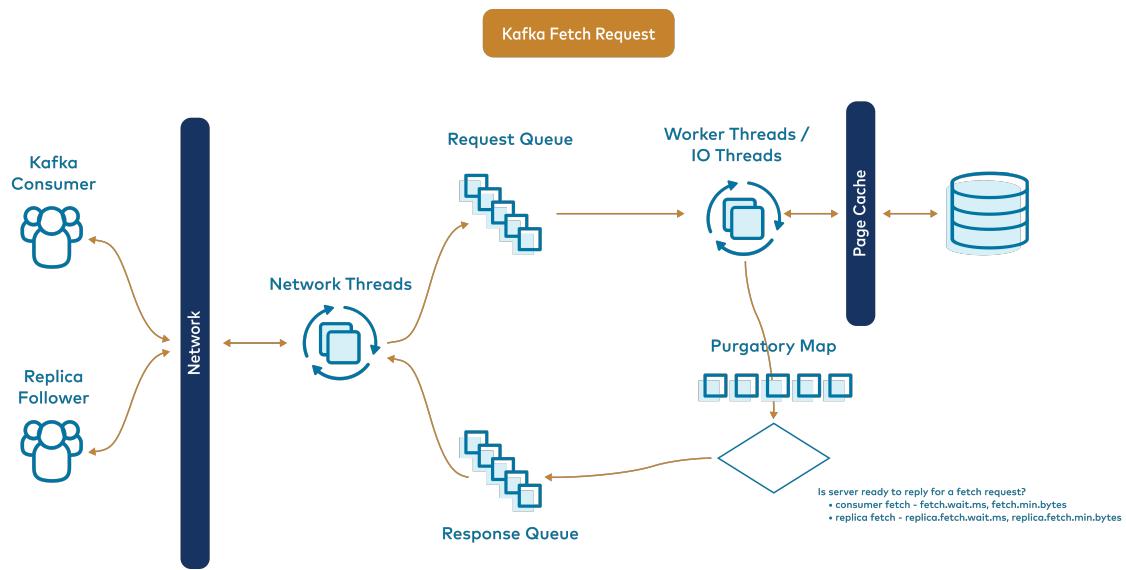
# Module Map



- Terminology Review
- Producer Performance
-  Hands-on Lab: **Exploring Producer Performance**
- Broker Performance
- Broker Failures and Recovery Time
- Load Balancing Consumption
-  Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance ... 
- Performance Testing
-  Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# Anatomy of a Fetch Request on a Broker



Note that this diagram is very similar to the one for Produce requests. All the structures are the same except for Purgatory - there are separate Purgatories for Produce and Fetch (Consume) requests.

# Tuning Consumer Performance

- High throughput:
  - Large `fetch.min.bytes` (Default: 1)
  - Reasonable `fetch.wait.max.ms` (Default: 500)
- Low latency:
  - `fetch.min.bytes=1`

---

With the `fetch.min.bytes` Consumer property, the Broker waits until this amount of data accumulates before sending a response to the Consumer. The Broker will not wait longer than `fetch.max.wait.ms`, however.

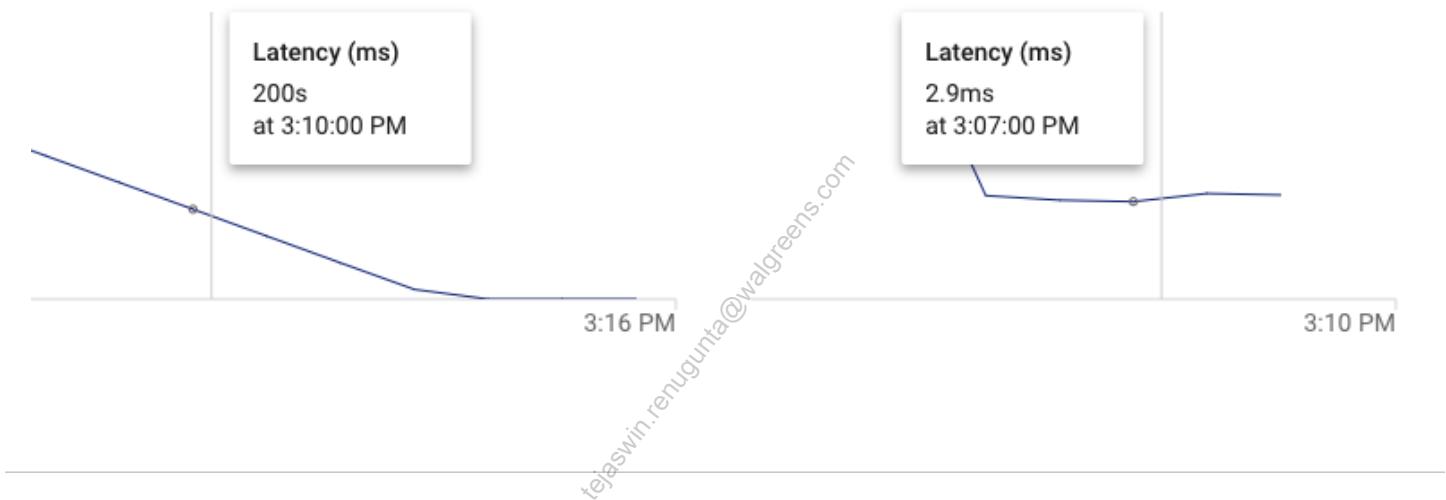
The Consumer can be tuned for real-time or batch behavior, just like the Producer. As with the Producer, the thresholds to choose the behavior are based on time (`fetch.max.wait.ms`) and size (`fetch.min.bytes`). The default behavior is real-time, as set by `fetch.min.bytes=1`. These settings are analogous to the Producer's `linger.ms` and `batch.size` settings.



As of Apache Kafka 2.5, there is a related broker-side configuration, `fetch.max.bytes`. The effective maximum size of any fetch request will be the minimum of `fetch.min.bytes` and this value. The default value for `fetch.max.bytes` is 55 megabytes. Fetch requests from replicas will also be affected by the `fetch.max.bytes` limit.

# Monitoring Consumer Performance

- Proactively monitor for slow Consumers
- `kafka.consumer:type=consumer-fetch-manager-metric,client-id=XXX`
  - `records-lag-max` and `fetch-rate`
    - Ideal: `records-lag-max = 0` and `fetch-rate > 0`
  - `records-consumed-rate` and `bytes-consumed-rate`
- Confluent Control Center can show slow, lagging Consumers (left) compared to good Consumers (right)



The `records-lag-max` metric calculates lag by comparing the offset most recently seen by the Consumer to the most recent offset in the log. This metric is important for real-time consumer applications where the Consumer should be processing the newest messages with as low latency as possible.

# Ensure High Performance with Quotas

- High volume clients can result in:
  - Monopolizing Broker resources
  - Network saturation
  - Denial of Service (DoS) other Producers and Consumers
  - DoS Brokers themselves
- Use **quotas** to throttle clients or groups of clients from overloading a Broker
  - Quotas are per-Broker, not cluster wide

---

Controlling the amount of bandwidth allowed for each client can be important if your environment has limited network resources. However, larger environments can also benefit since bandwidth throttling can be used for QoS style control over your cluster.

tejaswin.renugunta@walgreens.com

# How Quotas Work

- Quotas can be applied to:
  - Client-id: logical group of clients, identified by the same `client.id`
  - User: authenticated user principal
  - User and client-id pair: group of clients belonging to a user
- If quota exceeded, Broker will:
  1. Compute a delay time for the client
  2. Instruct client to not send more requests during delay
  3. Mute client channel so its requests are not processed during delay

---

"User" can also be a grouping of unauthenticated users chosen by the Broker using a configurable `PrincipalBuilder` and is currently used for ACLs.

Prior to AK 2.0, throttling was enforced by delaying the Broker response to the client when the quota was exceeded. For more information, see  
[https://kafka.apache.org/documentation/#design\\_quotasenforcement](https://kafka.apache.org/documentation/#design_quotasenforcement).

tejaswin.renugunta@walgreen.com

# Configure Quotas (1)

- Quotas can be defined by network bandwidth or request rate:
  - Network bandwidth: `producer_byte_rate`, `consumer_byte_rate`
  - Request rate: `request_percentage` (percentage of time a client can utilize the request handler I/O threads and network threads)
- Network bandwidth quota defaults, for example, to 1KBps

```
$ kafka-configs --bootstrap-server broker_host:9092 \
--alter \
--add-config 'producer_byte_rate=1024,consumer_byte_rate=1024' \
--client-defaults
```

---

The best practice is create a cluster-wide default quota and then adjust as necessary for specific user/client-id combinations. The examples on the slide illustrate how this could be implemented.

Request rate quotas were introduced in Kafka 0.11

# Configure Quotas (2)

- Request rate quota override for a specific client-id, user, or user and client-id pair

```
$ kafka-configs --bootstrap-server broker_host:9092 \
  --alter \
  --add-config 'request_percentage=50' \
  --client clientA \
  --user user1
```

- To describe the quota for a specific user and client-id pair

```
$ kafka-configs --bootstrap-server broker_host:9092 \
  --describe \
  --client clientA \
  --user user1
```

---

Quotas are typically configured with over-subscription. Total allocated quota for all clients is larger than capacity, but not all clients will go beyond the quota at the same time

# Monitoring Quota Metrics - Broker

```
kafka.server:type={Produce\|Fetch},client-id=([-.\w]+)
```

- Attribute **throttle-time** indicates the amount of time in milliseconds the client-id was throttled (0 if not throttled)
- Attribute **byte-rate** indicates the data produce/consume rate of the client in bytes/second

tejaswin.renugunta@walgreens.com

# Monitoring Quota Metrics - Producer

```
kafka.producer:type=producer-topic-metrics,client-id=(-.\w)+
```

- Attributes `produce-throttle-time-max` and `produce-throttle-time-avg`: Maximum and average times in milliseconds a request has been throttled

---

Checking quota metrics periodically is recommended. If clients are showing high throttle times, investigate why. In some cases, the quota may have been underestimated for a specific application and may need to be increased.

tejaswin.renugunta@walgreens.com

# Monitoring Quota Metrics - Consumer

```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.\w]+)
```

- Attributes `fetch-throttle-time-max` and `fetch-throttle-time-avg`: Maximum and average times in milliseconds a request has been throttled

tejaswin.renugunta@walgreens.com

# Discussion

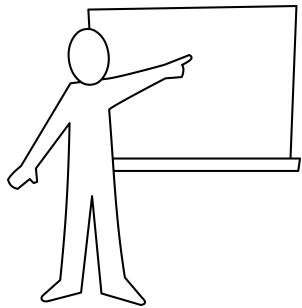
- You get a call from a customer who says one of their newly-written Kafka Consumers is slow. What do you do to investigate the problem?
- 

Give students several minutes to think about their response. Ask students to consult with a peer before engaging in whole-class discussion.

Here are some ideas to bring up during discussion if not mentioned by students:

- Is this a Broker or a Consumer issue?
- You might run `kafka-consumer-perf-test` on the same Topic. If the performance is equally bad, this could indicate a Broker-side issue
- Look for Broker-side metrics, each subset of the server's "TotalTime" during requests. Look for anomalies
- If the other Consumers are fast, with just this one being slow, it may be a localized Consumer-side issue
  - It may be that the application logic is slow; see if there is any instrumentation in the application showing the amount of time taken to process each message
- Another possibility is Garbage Collection
- Check that `fetch.min.bytes` and `fetch.max.wait` have not been changed
  - The default configuration values for the Consumer are typically good for consuming high throughput
- Check configured quotas that may apply to that Consumer

# Module Map



- Terminology Review
- Producer Performance
-  Hands-on Lab: **Exploring Producer Performance**
- Broker Performance
- Broker Failures and Recovery Time
- Load Balancing Consumption
-  Hands-on Lab: **Modifying Partitions and Viewing Offsets**
- Consumption Performance
- Performance Testing ... 
-  Hands-on Lab: **Performance Tuning**

tejaswin.renugunta@walgreens.com

# Why Test Kafka Performance At All?

- Benchmarks establish baseline for performance
  - Broker and client benchmarks → capacity planning
  - Analyze effect of cluster/client changes against baseline

---

Establishing benchmarks is an important task for planning and troubleshooting. Unless you know the expected performance of your Brokers and clients, it is very difficult to size your environment and respond to performance issues. Without a baseline, you can't understand the impact of generating higher volumes of data, adding Producers, Consumers, etc.



Remember that not all environments are created equal! Your test/dev environment may have different performance characteristics than your production environment so make sure to test both.

tejaswin.renugunta@walgreens.com

# How to Test Performance

## kafka-producer-perf-test

- Run a single Producer on a single server
- Determine  $p$ : Producer throughput per Partition

## kafka-consumer-perf-test

- Run a single Consumer on a single server
- Determine  $c$ : Consumer throughput per Partition

---

Kafka comes with Producer and Consumer benchmarking tools. These are a great place to start, but it is also important to test instances of your actual Producers and Consumers in a controlled way that avoids cluster bottlenecks.

Producer throughput is typically easy to benchmark. Run the application at a steady state to determine how much data the system can pass.

Consumers can be more challenging. How much data can be processed depends on many factors, e.g., processing time per message, number of messages fetched per poll, message size (maximum or average). Test with as many combinations of these types of variables as is reasonable for your environment to get an accurate throughput number for your Consumer.

tejaswin.reddy@algens.com

# Measuring Throughput

- All Topics bytes/messages in (Meter)

```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec  
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec  
kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesInPerSec
```

- All Topics bytes out (Meter)

```
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec  
kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesOutPerSec
```

- Confluent Control Center provides per-Broker and per-Topic throughput metrics

---

Kafka provides inbound and outbound metrics for bytes and messages on a Broker. These data points are also available on a per-Topic basis.

**LeaderCount** and **PartitionCount** are critical to analyze performance, i.e., make sure leaders are balanced across the Brokers.

Version notes:

- Kafka 0.11.0:
  - **ReplicationBytesInPerSec** and **ReplicationBytesOutPerSec** were added
  - **BytesOutPerSec** now only tracks Consumer traffic (i.e., does not include replication traffic)

# Number of Partitions and Client Properties

- Ideal number of partitions:  $\max(t/p, t/c)$ 
    - $t$ : target throughput
    - $p$ : Producer throughput per Partition
    - $c$ : Consumer throughput per Partition
- 

- Vary Producer properties:
  - Replication factor
  - Message size
  - In flight requests per connection  
(`max.in.flight.requests.per.connection`)
  - Batch size (`batch.size`)
  - Batch wait time (`linger.ms`)
- Vary Consumer properties:
  - Fetch size (`fetch.min.bytes`)
  - Fetch wait time (`fetch.max.wait.ms`)

As mentioned previously, the limiting factor is likely to be the Consumers, so the number of Partitions will likely be  $t/c$ . Topics should be sized so that Consumers can keep up with the throughput from a physical (NIC speed) and computational (processing time per poll) standpoint.

feswin.renugunta@walgreens.com

# Improving Throughput With More Partitions

- More Partitions → higher throughput
  - Rule of thumb for maximums:
    - Up to 4,000 Partitions per Broker
    - Up to 200,000 Partitions per cluster
- 

One of the most common questions asked about Kafka is "How many Partitions should my Topic have?"

There is no simple answer. More Partitions generally means higher throughput for Consumers (assuming you have enough Consumers to assign to all the Partitions). For Producers of keyed messages, there is keyspace to consider. If there are only 100 unique keys, and all messages of the same key are guaranteed to land on the same Partition, then having more than 100 Partitions doesn't make sense.

There are downsides to arbitrarily large Partition counts that we will discuss on the next slide. The limits shown on the slide (<4K Partitions/Broker, <200 K Partitions per cluster) are *maximums*. Most environments are well below these values (typically in the 1000-1500 range or less per Broker).

The maximums for Partitions were updated in AK 2.0. Previously the maximums were listed as 2-4K Partitions per Broker and 10s of thousands of Partitions per cluster.

# Downside to More Partitions

- More open file handles
- Longer leader elections → more downtime after Broker failure
- Higher latency due to replication
- More client memory (buffering per Partition)



When producing keyed messages: avoid unbalanced key utilization. This leads to "hot Partitions."

- Downsides with too many Partitions:
  - You need more open file handles: More Partitions means more directories and segment files on disk.
  - Availability issue: Planned failures move Leaders off of a Broker one at a time, with minimal downtime per Partition. In a hard failure all the leaders are immediately unavailable. The Controller needs to detect the failure and choose other leaders, but since this happens one at a time, so it can take a long time for them all to be available again. The first Partition will be offline for significantly less time than the nth Partition. Additionally, if the Controller itself fails, the first thing that has to happen is to failover the Controller. The new Controller needs to initialize by reading a lot of metadata, i.e., the metadata for all Partitions in the cluster. The more Partitions, the longer this recovery takes.
  - Latency impact: only really matters when you're talking about millisecond latency. For the message to be seen by a Consumer it must be committed. The Broker replicates data from the leader with a single thread, resulting in overhead per Partition. If you have 1000 Partitions the overhead is about 20 milliseconds.
  - Client Memory: Both the Producer and Consumer buffer per-Partition. Increasing the number of Partitions would increase the memory requirements on the clients.
  - Resizing: A Topic can be expanded if it was created with too few Partitions. Topics cannot reduce the number of Partitions they contain.

# Hands-On Lab

- You will monitor and tune Consumer performance.
- Please refer to **Lab 05 Optimizing Kafka's Performance** in the Exercise Book:
  - a. **Performance Tuning**



tejaswin.renugunta@walgreens.com

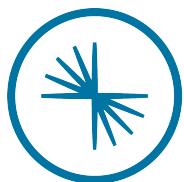
# Module Review



- Kafka provides configuration properties to performance tune for high throughput and low latency
- The number of Partitions can impact the performance
- Consumer Groups allow consumption load balancing
- Quotas can prevent a single client from using up all resources on the Brokers
- Kafka provides a lot of metrics for monitoring performance

tejaswin.renugunta@walgreens.com

# 06: Kafka Security



CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Providing Durability
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance
6. Kafka Security ... ↙
7. Data Pipelines with Kafka Connect
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

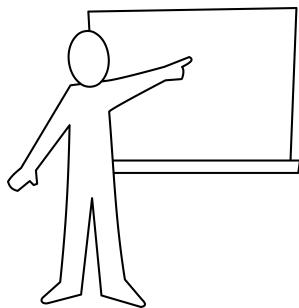


After this module you will be able to:

- Configure encryption, authentication, and authorization on a cluster
- Discuss tradeoffs of various security configurations
- Migrate from an insecure to a secure cluster

tejaswin.renugunta@walgreens.com

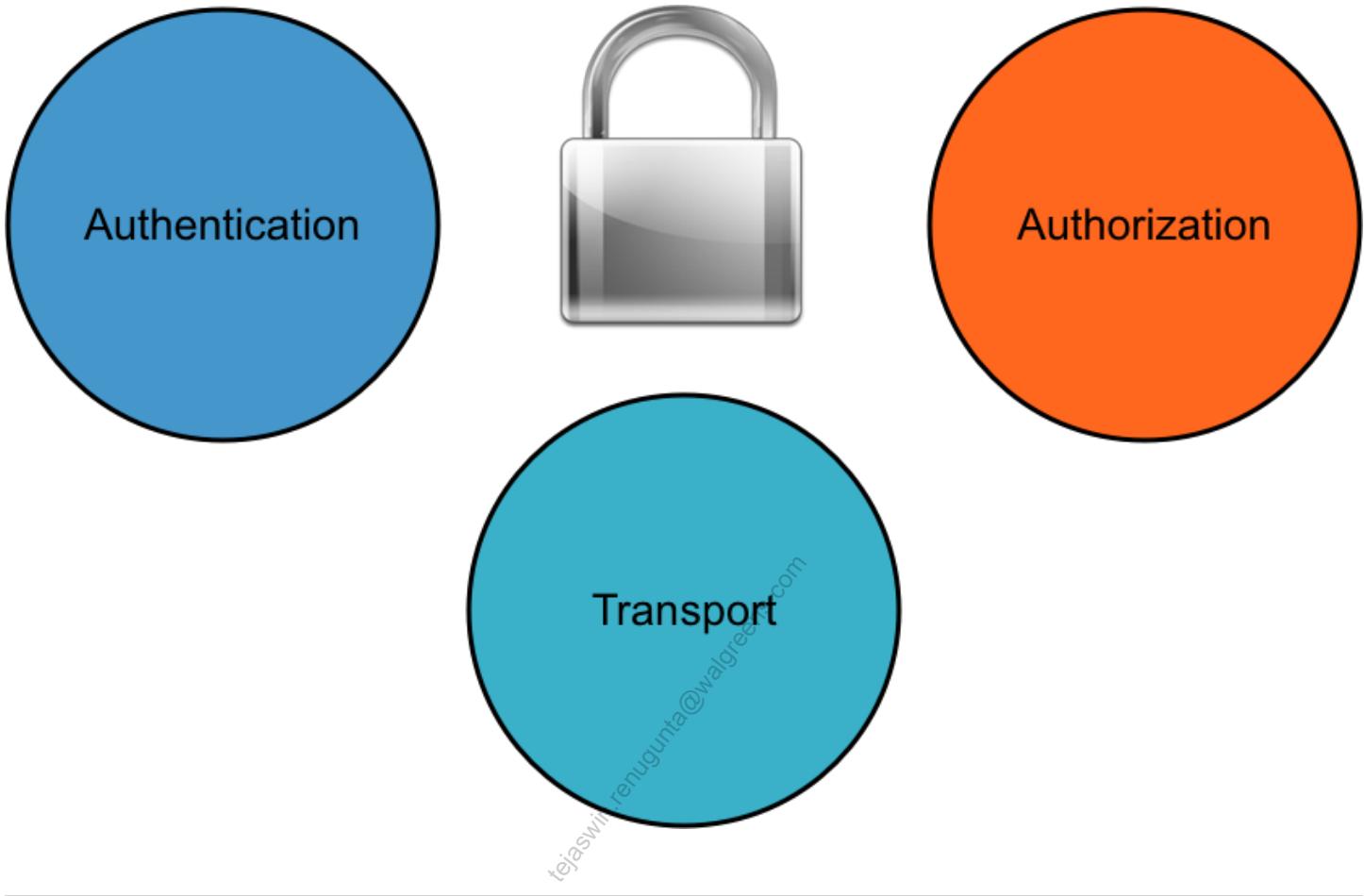
# Module Map



- Overview ... ←
- SSL for Encryption and Authentication
- SASL for Authentication
- Authorization
- Securing the Whole Environment
- Migration to a Secure Cluster
- Security - Confluent Cloud
-  Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# Security Overview



## Authentication:

- One party verifies the identity of another party.
  - Example: The bank is giving a loan to a customer. The customer provides a valid passport for identification. The bank trusts the government as an authority to confirm that the customer is who they claim to be. To make this fully analogous, the bank would be able to compute a hash of the passport and check with the government ("certificate authority") that the passport hasn't been altered.

## Authorization:

- Given that identity has been established, the authorizing party decides whether the other party's request should be granted.
  - Example: Now that the bank believes the customer's identity, they decide whether the customer's request for a loan should be granted.

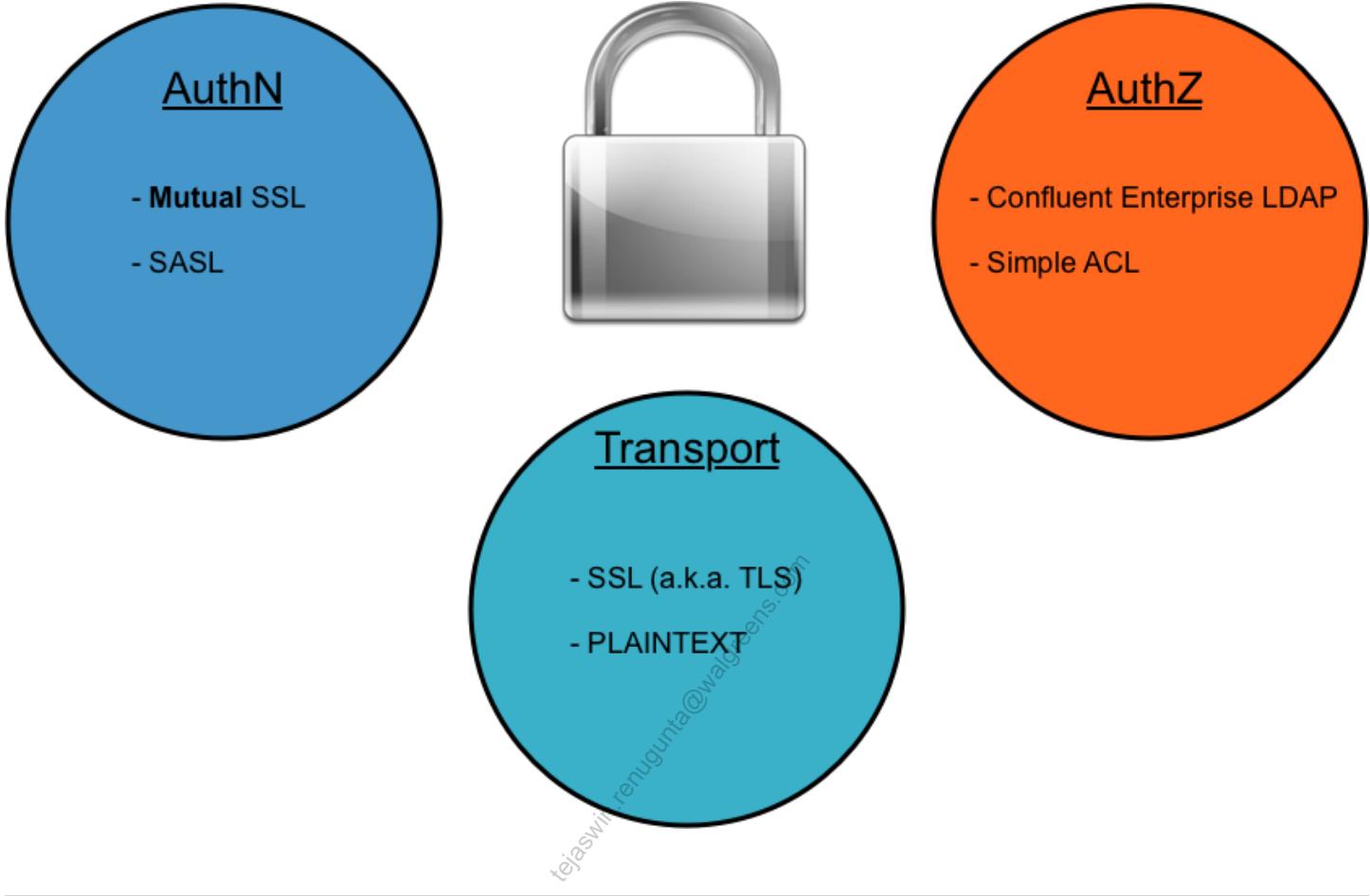
## Transport Security:

- Can an uninvited guest listen to the information being transferred?
  - Example: If the bank and the customer are exchanging all of this information in plain view and speaking out loud, an eavesdropper can gather quite a lot of information. To prevent this, the bank and the customer could agree to send coded messages to each other that only they know how to read.

In this module, students will consider how kafka clients authenticate with Brokers, how Brokers authenticate with each other, how Brokers authenticate with clients, and how to use an authorizer plugin to enforce access rules on the cluster.

tejaswin.renugunta@walgreens.com

# Kafka Security Features



- Transport
  - Data can be transported without encryption as **PLAINTEXT**, or with encryption using SSL (Secure Socket Layer).
- Authentication
  - The SSL protocol actually authenticates a server with a client by nature (one-way authentication). That means we can use SSL to authenticate the client with the server as well, which is called mutual SSL.
  - Kafka supports several authentication mechanisms which will be discussed in various levels of depth in later slides. These mechanisms use the SASL protocol (Simple Authentication Security Layer).
- Authorization
  - Kafka comes with a simple authorizer plugin to allow or deny access to cluster resources with Access Control Lists (ACLs).

- Confluent Enterprise offers an LDAP authorizer plugin so that an enterprise can easily integrate Kafka into an existing LDAP infrastructure

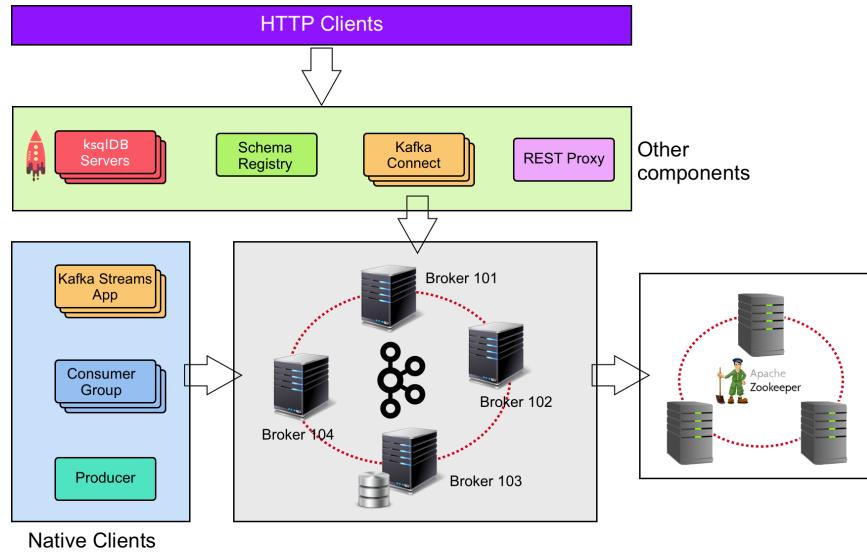


Although the documentation and Kafka configuration settings refer to SSL, the protocol used is actually TLS (Transport Layer Security). The SSL protocol was deprecated in favor of TLS in June 2015. Kafka and Java refer to transport layer security as SSL, and these materials will follow with that convention.

TLS 1.3 is the default TLS protocol when using Java 11 or higher, and TLS 1.2 is the default for earlier Java versions. TLS 1.0 and 1.1 are disabled by default due to known security vulnerabilities, though users can still enable them if required.

tejaswin.renugunta@walgreens.com

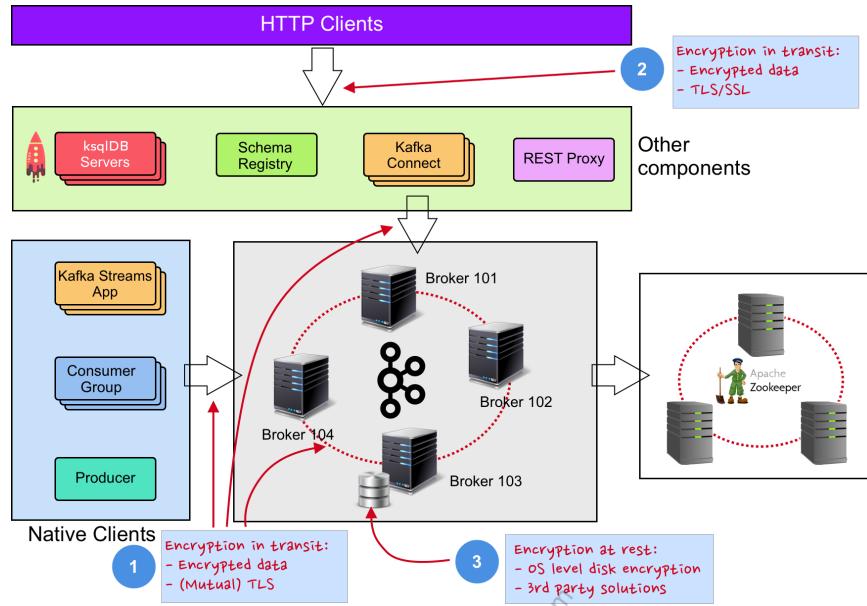
# Security - Architecture



Here we show an overview over a potential Confluent real-time streaming platform. We have the following components:

- At the center lies the Kafka cluster with its 1 to many Brokers
- Right of it we have the ZooKeeper ensemble, typically 3 to 5 instances for high availability
- On the left hand side we have the so called native Kafka clients: Producers, Consumer groups, Kafka Streams applications and ksqlDB Server clusters
- On top we have, what I call, middle ware: the Confluent Schema Registry, Kafka Connect and the Confluent REST Proxy
- Finally at the very top, in purple we have the HTTP clients that access the middle ware via REST API

# Security - Encryption at Rest & in Transit



The Kafka cluster, including the ZooKeeper ensemble can be secured as follows:

## 1. Encryption **in-transit** (or **in-flight**) is mainly achieved in the following 2 ways:

- the Producer encrypts the data before sending it to the Broker. The Consumer decrypts this data
- using (mutual) TLS

The above applies for both situations:

- client → Broker
- Broker → Broker

## 2. Encryption between HTTP clients and Confluent Platform services such as Schema Registry happens in the following ways

- Using TLS/SSL
- Using the Confluent Enterprise security plugin

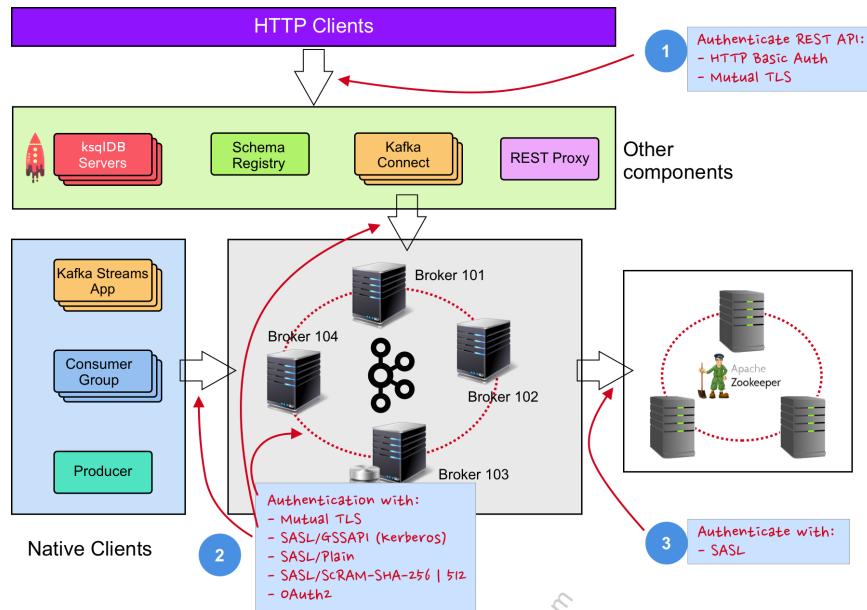
## 3. Encryption at rest is implemented either by using OS level disk partition encryption or by using 3rd party services such as the Vorometric Data Security manager



Although using encryption/decryption at the client level normally inhibits us from working with ksqlDB, an engineer from Confluent created UDFs which support at-rest encryption/decryption at the field level in a proof of concept (POC), showing that it can be done.

tejaswin.renugunta@walgreens.com

# Authentication



Confluent Control Center, ksqlDB, Schema Registry, Kafka Connect, and REST Proxy all have REST APIs and can all be configured with basic username/password authentication over HTTP (plaintext over the wire). They should be further configured so requests are encrypted over the wire (HTTPS). This module focuses on security with Kafka clients, Brokers, and ZooKeeper, rather than REST API clients, so we only briefly mention REST API security here.

+ NOTE: Confluent Control Center is not pictured, but it is also a REST API server for the monitoring UI.

Here are further resources on securing the various REST API services:

- HTTP Basic AuthN for all REST API services:  
<https://docs.confluent.io/current/security/basic-auth.html>
- Confluent Control Center HTTPS: <https://docs.confluent.io/current/control-center/security/authentication.html#ui-https>
- Kafka Connect HTTPS:  
<https://docs.confluent.io/current/kafka/encryption.html#encryption-ssl-rest>
- Schema Registry HTTPS: <https://docs.confluent.io/current/schema-registry/security.html#additional-configurations-for-https>
- REST Proxy HTTPS: <https://docs.confluent.io/current/kafka/encryption.html#rest-proxy>

- KSQL HTTPS: <https://docs.confluent.io/current/ksql/docs/installation/server-config/security.html#configuring-ksql-for-https>
2. Kafka clients (including the REST API **servers** just mentioned) can authenticate with the Kafka cluster with SASL or Mutual SSL.
- SASL GSSAPI is the authentication mechanism used by Kerberos, a popular enterprise-wide security technology. This mechanism will get most attention in this module, but other SASL mechanisms are available as well.
3. Kafka Brokers authenticate with ZooKeeper. ZooKeeper supports SASL authentication mechanisms.



ZooKeeper version 3.4 does not support SSL

tejaswin.renugunta@walgreens.com

# Broker Ports for Security

- Plain text (no wire encryption, no authentication)

```
listeners=PLAINTEXT://kafka-1:9092
```

- SSL (wire encryption, authentication)

```
listeners=SSL://kafka-1:9093
```

- SASL (authentication)

```
listeners=SASL_PLAINTEXT://kafka-  
1:9094
```

- SSL + SASL (SSL for wire encryption, SASL for authentication)

```
listeners=SASL_SSL://kafka-1:9095
```

- Clients choose **only one** port to use



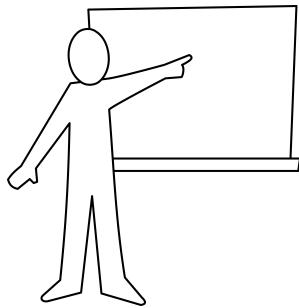
If clients connect from an outside network, make sure to set `advertised.listeners` in addition to `listeners` so that clients can discover the Brokers

Kafka Brokers can listen on multiple ports at the same time so that different clients can be authenticated appropriately. This is done using a comma separated list for the `listeners` property. Clients must choose one port; they cannot failover automatically if their preferred login method is unavailable.

SASL usually refers to Kerberos but can also be SCRAM or Plain for the username/password storage.

This may be a good time to ask students about the difference between the `listeners` and `advertised.listeners` properties. If there is confusion on this point, explain that `listeners` is the interface for connecting to a kafka Broker locally, whereas `advertised.listeners` is the interface that is advertised for clients outside a local network use to connect to the Broker. For an illustrated explanation of listeners, see this blog post: <https://rmoff.net/2018/08/02/kafka-listeners-explained/>.

# Module Map



- Overview
- SSL for Encryption and Authentication ... ↵
- SASL for Authentication
- Authorization
- Securing the Whole Environment
- Migration to a Secure Cluster
- Security - Confluent Cloud
- 🔑 Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# Why is SSL Useful?

- Organization or legal requirements
- One-way authentication
  - Secure wire transfer using encryption
  - Client knows identity of Broker
  - Use case: Wire encryption during cross-data center mirroring
- Two-way authentication with **Mutual SSL**
  - Broker knows the identity of the client as well
  - Use case: authorization based on SSL principals
- Easy to get started
  - Just requires configuring the client and server
  - No extra servers needed, but can integrate with enterprise

---

SSL can be used with one-way or two-way authentication.

One-way authentication is similar to what you have used with secure websites. The client verifies that the server is trusted before exchanging data. However, that only guarantees that the server is good; this does not prevent unauthorized access to Brokers and Topics. Two-way authentication requires that the client identity be verified as well.



It is outside the scope of the course to provide in-depth configurations for SSL. The materials address configurations specific to the Kafka environment.

# SSL Data Transfer

- After an initial handshake, data is encrypted with the agreed-upon encryption algorithm
- There is overhead involved with data encryption:
  - Overhead to encrypt/decrypt the data
  - Can no longer use zero-copy data transfer to the Consumer
  - SSL overhead will increase

```
kafka.network:type=RequestMetrics,name=ResponseSendTimeMs
```

---

In an SSL connection, the Producer encrypts the messages before sending to the Brokers. The Brokers decrypt the messages for local storage and then re-encrypt them before sending to the Consumers, who then have to decrypt the messages again.

All of this processing will affect end-to-end performance and CPU utilization on all components of the Kafka architecture.

tejaswin.renugunta@walgreens.com

# SSL Performance Impact

- Performance was measured on Amazon EC2 r3.xlarge instances

	Throughput(MB/s)	CPU on client	CPU on Broker
Producer (plaintext)	83	12%	30%
Producer (SSL)	69	28%	48%
Consumer (plaintext)	83	8%	2%
Consumer (SSL)	69	27%	24%

These performance numbers are specific to the described environment - actual numbers will vary.

The increase in CPU utilization for the Consumer connection seems excessive but that is just because those activities require so little CPU in normal circumstances.

	There are significant performance improvements for versions of Kafka that use Java 9 or above. Confluent Platform 5.2 and above support Java 11.
--	--

# Data at Rest Encryption

- "Data at rest encryption" refers to encrypting data stored on a hard drive that is currently not moving through the network
  - Options for encrypting data at rest:
    1. Linux OS encryption utilities that encrypt full disk or disk partitions (e.g. LUKS)
    2. Producers encrypt messages before sending to Kafka, and Consumers decrypt after consuming
  - Consider running Brokers on a machine with an encrypted filesystem or encrypted RAID controller
- 

SSL only provides wire encryption. Kafka itself is not capable of encrypting data stored on disk.

One advantage of pushing encryption to the Kafka client layer is that Brokers can function without the performance impact of SSL.

tejaswin.renugunta@walgreens.com

# SSL: Keystores and Truststores

Kafka client truststore.jks



Kafka Broker keystore.jks



- Client truststore:

- CA certificate used to verify signature on Broker cert
- If signature is valid, Broker is **trusted**

- Broker keystore:

- Uses private key to create **certificate**
- Cert must be signed by Certificate Authority (CA)
- Cert includes public key, which client will use to establish secure connection

Java uses **.jks** (Java Key Store) files to hold information used during SSL handshake. The client truststore and Broker keystore are what must be configured by Kafka cluster administrators, so it is not necessary to go into deeper detail about the SSL/TLS protocol explicitly. However, a short summary is given here for reference.

Short summary of SSL for reference: One important concept throughout is that a public key can be used to encrypt information in such a way that only its associated private key can decrypt; and a private key can also be used to encrypt information in such a way that only the public key can decrypt. The Broker creates a public/private key-pair. The private key is used to generate a certificate which contains its public key. The certificate must be signed by a certificate authority that is trusted by the client. The CA uses its own private key to sign certificates. During the SSL handshake, the Broker sends its certificate, including its public key, to the client. The client verifies the CA signature by using the CA's certificate (which includes the CA's public key). If the certificate is trusted, then the client creates a shared key that the Broker and client will use during the session (symmetric encryption). In order to securely send the shared key back to the Broker, the client uses the Broker's public key to encrypt the shared key. Only the Broker's private key can decrypt this information, so the Broker is able to recover the shared key. Now that Broker and client each have the same key, they use that shared key to encrypt and decrypt subsequent communication in the session. For a simple, illustrated video on the SSL encryption process, see <https://www.youtube.com/watch?v=4nGrOpoOCuc>.

# Preparing SSL

1. Generate certificate (X.509) in Broker **keystore**
2. Generate Certificate Authority (CA) for signing
3. Sign Broker certificate with CA
4. Import signed certificate and CA certificate to Broker **keystore**
5. Import CA certificate to client **truststore**



Mutual SSL: generate client **keystore** and corresponding Broker **truststore** in the same way.

---

Kafka leverages standard SSL procedures - there are no Kafka-specific steps that have to be done to create the certificates. For more information on configuring SSL transport encryption, see <https://docs.confluent.io/current/kafka/encryption.html>.

tejaswin.renugunta@walgreens.com

# SSL Everywhere! (1)

- Client and Broker `*.properties` Files:

```
ssl.keystore.location = /var/private/ssl/kafka.server.keystore.jks  
ssl.keystore.password = password-to-keystore-file  
ssl.key.password = password-to-private-key  
ssl.truststore.location = /var/private/ssl/kafka.server.truststore.jks  
ssl.truststore.password = password-to-truststore-file
```

- Brokers:

```
listeners = SSL://<host>:<port>  
ssl.client.auth = required  
inter.broker.listener.name = SSL
```

- Client:

```
security.protocol = SSL
```

These configurations demonstrate mutual SSL, which means client authentication is required. In addition to the configurations on the slide, remember to also change the port number for the entries in the `bootstrap.servers` configuration on the clients. For more detailed information on two-way SSL, see [https://docs.confluent.io/current/kafka/authentication\\_ssl.html#kafka-ssl-authentication](https://docs.confluent.io/current/kafka/authentication_ssl.html#kafka-ssl-authentication).

 Setting `inter.broker.listener.name` to `SSL` ensures that communication between Brokers is also SSL encrypted, which means each Broker's truststore must trust the certificates of each other Broker. To do this, you can use a single Certificate Authority to sign all Broker certificates. To reduce the possibility of a client trying to impersonate a Broker when doing mutual SSL, you could use a separate Certificate Authority to sign client certificates.

# SSL Everywhere! (2)

Discussion Questions:

- How could you secure the clear credentials that are stored in the `*.properties` files?
  - What configurations would you change to do one-way SSL (SSL from Broker to client only) and plaintext between Brokers?
- 

Here are some ideas to bring up if not mentioned by students:

- How could you secure the clear credentials that are stored in the `*.properties` files?
  - set OS level permission restrictions on the files
  - Use a disk encryption tool
- What configurations would you change to do one-way SSL (SSL from Broker to client only) and plaintext between Brokers?
  - Create a new `PLAINTEXT` listener port on the Broker for inter-Broker communication
  - Change `inter.broker.listener.name` to `PLAINTEXT` on the Broker
  - Client `*.properties` file no longer needs keystore configurations (only truststore)
  - Delete `ssl.client.auth = required`



Upcoming versions of Confluent Platform will provide command line tools for securing passwords so they are not plain text on `*.properties` files.

# SSL Principal Name

- By default, Principal Name is the distinguished name of the certificate

```
CN=host1.example.com,OU=organizational  
unit,O=organization,L=location,ST=state,C=country
```

- Can be customized through `principal.builder.class`
  - Has access to X509 Certificate
  - Makes setting the Broker principal and application principal convenient

---

As of AK 2.2.0, it is possible to use regular expressions to extract a custom principal from the distinguished string: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-371%3A+Add+a+configuration+to+build+custom+SSL+principal+name>.

tejaswin.renugunta@walgreens.com

# Troubleshooting SSL

- To troubleshoot SSL issues
  - Verify all Broker security configurations
  - Verify client security configuration
  - On the Broker, check that the following command returns a certificate:

```
$ openssl s_client -connect <broker>:<port> -tls1
```

- Enable SSL debugging using the `KAFKA_OPTS` environment variable
  - The output can be verbose but it will show the SSL handshake sequence, etc.
  - If you are debugging on the Broker, you must restart the Broker

```
$ export KAFKA_OPTS="-Djavax.net.debug=ssl $KAFKA_OPTS"
```

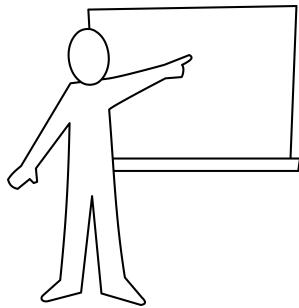
---

SSL debugs ('-Djavax.net.debug=ssl') can be verbose but helpful. For example, if two-way SSL authentication is enabled with `ssl.client.auth=required` but client has no keystore configured, then the debugs would show

```
Warning: no suitable certificate found - continuing without client authentication
*** Certificate chain
<Empty>
```

This may be a good time to check how familiar students are with the `KAFKA_OPTS` variable. If students are unclear, remind them that this is the variable used to pass configurations to the JVM. This variable is invoked when Kafka starts (e.g. `kafka-server-start`).

# Module Map



- Overview
- SSL for Encryption and Authentication
- SASL for Authentication ... ↵
- Authorization
- Securing the Whole Environment
- Migration to a Secure Cluster
- Security - Confluent Cloud
- 🔑 Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# SASL for Authentication (1)

- SASL: Simple Authentication and Security Layer
    - Challenge/response protocols
    - Server issues challenge; client sends response
    - Continues until server is satisfied
  - All of the SASL authentication methods send data over the wire in **PLAINTEXT** by default, but SASL authentication can (should) be combined with **SSL** transport encryption.
- 

SASL can be used with either encryption (**SSL**) or not (**PLAINTEXT**). Since usernames and passwords are sent in the clear for some SASL mechanisms, it is highly recommended in production environments that the connections be encrypted. However, encryption will have the same performance issues as described in the SSL slides.

tejaswin.renugunta@walgreens.com

# SASL for Authentication (2)

- SASL supports different mechanisms
  - GSSAPI: Kerberos
  - SCRAM-SHA-256, SCRAM-SHA-512: “salted” and hashed passwords
    - SCRAM (Salted Challenge Response Authentication Mechanism)
    - Can use username/password stored in ZK or delegation token (OAuth 2)
  - PLAIN: cleartext username/password
  - OAUTHBEARER: authentication tokens

---

SASL provides multiple authentication methods.

Kerberos is frequently used because it enables single sign-on. However, as with SSL, there are no Kafka-specific changes that need to be made to the servers. It is beyond the scope of this course to set up the Kerberos environment.

The SCRAM implementation is described in [IETF RFC 5802](#). Other SCRAM mechanisms like SHA-224 and SHA-384 are not supported at this time. Strong hash functions combined with strong passwords and high iteration counts protect against brute force attacks if Zookeeper security is compromised. Support for delegation tokens was added in Kafka 2.0.

PLAIN is the easiest of the SASL options but is not generally used in production environments.

	<p><b>PLAIN</b> is an authentication mechanism, whereas <b>PLAINTEXT</b> is transporting data without encryption. It's possible to use SASL PLAIN with <b>SSL</b> or <b>PLAINTEXT</b>, just like any other SASL mechanism.</p>
---	--

OAUTHBEARER (introduced in Kafka 2.0/Confluent 5.0) is a self-service token based authentication method. It uses unsecured JSON web tokens by default and should be considered non-production without additional configuration.

These materials will only discuss SASL SCRAM and SASL GSSAPI in more detail.

# Using SASL SCRAM

- SCRAM should be used with SSL for secure authentication
- ZooKeeper is used for the credential store
  - Create credentials for Brokers and clients
  - Credentials must be created before Brokers are started
  - ZooKeeper should be secure and on a private network

---

SCRAM is perfectly acceptable for smaller teams with less stringent security needs. This is a fairly common scenario. If ZooKeeper is secure and on a private network, SCRAM can even be used for production systems.

tejaswin.renugunta@walgreens.com

# Configuring SASL SCRAM Credentials

- Create credentials for inter-Broker communication (user "admin")

```
$ kafka-configs --bootstrap-server broker_host:port \
--alter \
--add-config \
'SCRAM-SHA-256=[password=admin-secret],SCRAM-SHA-512=[password=admin-
secret]' \
--user admin
```

- Create credentials for Broker-client communication (e.g. user "alice")

```
$ kafka-configs --bootstrap-server broker_host:port \
--alter \
--add-config \
'SCRAM-SHA-256=[password=alice-secret],SCRAM-SHA-512=[password=alice-
secret]' \
--user alice
```

Client configurations are created and managed with the `kafka-configs` command. These changes do not require a reboot.



While plaintext passwords are not sent over the wire, they are available on the host's `.bash_history` file. Take precautions to restrict access to command history and clear it regularly (e.g. `history -c`).

# Configuring SASL Using a JAAS File (Broker)

- JAAS (Java Authentication and Authorization Service) can be included in `*.properties` files on clients and Brokers using the `sasl.jaas.config` property

Broker (`server.properties`):

```
...
listeners=SASL_SSL://<host>:<port>
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256
sasl.enabled.mechanisms=SCRAM-SHA-256

listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
        username="admin" \
        password="admin-secret";
...
...
```

- 
- This example continues the configuration of SCRAM 256 for authentication and SSL for transport encryption, but these property settings are analogous for any SASL mechanism
  - Supports dynamic configuration on Brokers with `kafka-configs`
  - The property name must be prefixed with the listener prefix including the SASL mechanism, i.e.  
`listener.name.<listenerName>.<saslMechanism>.sasl.jaas.config`.
  - If multiple mechanisms are configured on a listener, configs must be provided for each mechanism using the listener and mechanism prefix. For example:

```
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
        username="admin" \
        password="admin-secret";
listener.name.sasl_ssl.gssapi.sasl.jaas.config= \
    com.sun.security.auth.module.Krb5LoginModule required \
        useKeyTab=true \
        storeKey=true \
        keyTab="/etc/security/keytabs/kafka_server.keytab" \
        principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
```

# Configuring SASL Using a JAAS File (Client)

Client configuration file:

```
...
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-256

sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
        username="alice" \
        password="alice-secret";
...
...
```

tejaswin.renugunta@walgreens.com

# Discussion questions:

- Why is `SASL_SSL` wire encryption recommended when using SASL SCRAM?
  - What are the tradeoffs of an environment where clients authenticate with SASL SCRAM over SSL, but inter-Broker communication is done over plaintext? How would you change these `*.properties` files for this situation?
  - Notice again that clear passwords are stored in these files. What would you do to ensure the security of these credentials?
- 

- Why is `SASL_SSL` wire encryption recommended when using SASL SCRAM?
  - While SCRAM does salt and hash passwords in transit, there is enough information in a SCRAM exchange to make individual passwords vulnerable to a dictionary or brute force attack if ZooKeeper is compromised (although the random salt does prevent a dictionary attack from cracking all passwords at once). Strong passwords, strong hash functions, and high iteration counts all work to minimize this as a viable attack vector.
- What are the tradeoffs of an environment where clients authenticate with SASL SCRAM over SSL, but inter-Broker communication is done over plaintext? How would you change these `*.properties` files for this situation?
  - If a Kafka cluster in a private network is considered "secure enough", then it may be worth it to only encrypt communication from clients outside the private network while allowing Brokers to communicate with each other over plaintext. This can reduce the performance impact that comes with TLS.
  - To enable this, we would have to create a new plaintext listener port on the Brokers for inter-Broker communication, e.g. `SASL_PLAINTEXT://<host>:<port>` and set `inter.broker.listener.name=SASL_PLAINTEXT`. The client configuration would be unchanged unless the client were also within the trusted network.
- Notice again that clear passwords are stored in these files. What would you do to ensure the security of these credentials?
  - These files should be configured with restricted file permissions.
  - Use disk level encryption.
  - There are unresolved KIPs that suggest ways to allow credentials to be passed from secure remote secret stores.

For more information on SCRAM and its security implications, see

[https://docs.confluent.io/current/kafka/authentication\\_sasl/authentication\\_sasl\\_scram.html#kafka-sasl-auth-scram](https://docs.confluent.io/current/kafka/authentication_sasl/authentication_sasl_scram.html#kafka-sasl-auth-scram)

# Using Delegation Tokens

- Use `kafka-delegation-tokens` tool complement to SASL SCRAM
- Typical steps for delegation token usage are:
  - User authenticates with the Kafka cluster and obtains a delegation token
  - User securely passes the delegation token to Kafka clients for authenticating with the Kafka cluster:

```
sasl.jaas.config = \  
    org.apache.kafka.common.security.scram.ScramLoginModule required \  
        username="tokenID123" \  
        password="lAYYSFmLs4bTjf+ltZ1LCHR/ZZFNA==" \  
        tokenauth="true";
```

- Token owner/renewer manually renews/expires the delegation tokens

---

Delegation tokens provide a lightweight authentication mechanism to complement SASL SCRAM. Delegation tokens are shared secrets between Kafka Brokers and clients. Delegation tokens will help processing frameworks to distribute the workload to available workers in a secure environment without the added cost of distributing Kerberos TGT/keytabs or keystores when 2-way SSL is used. See the documentation for more details: [https://docs.confluent.io/current/kafka/authentication\\_sasl/authentication\\_sasl\\_delegation.html#authentication-using-delegation-tokens](https://docs.confluent.io/current/kafka/authentication_sasl/authentication_sasl_delegation.html#authentication-using-delegation-tokens)

A master secret key is used to generate and verify tokens; a single value of `delegation.token.master.key` is configured on each Broker, which of course means this file needs to be diligently secured on each Broker.

Clients can obtain the delegation token using AdminClient APIs or using the `kafka-delegation-tokens` script.

Secret rotation is currently not supported, so the master key must be rotated manually:

1. Expire all existing tokens.
2. Rotate the master key with a rolling update of the cluster.
3. Generate and distribute new tokens.

# Important Broker Configuration Properties

- `delegation.token.master.key` (must be rotated manually)
- `delegation.token.expiry.time.ms` (Default: 1 Day)
- `delegation.token.max.lifetime.ms` (Default: 7 Days)

tejaswin.renugunta@walgreens.com

# Why Kerberos?

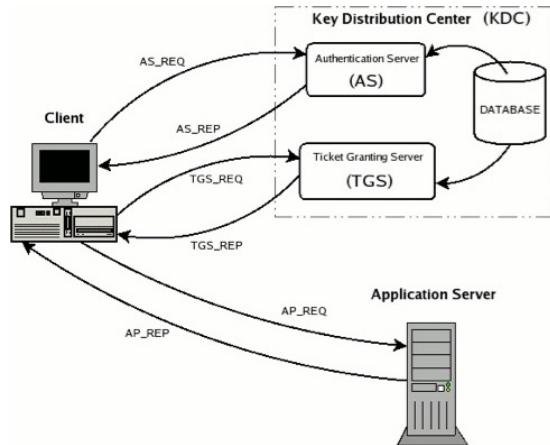
- Kerberos provides secure single sign-on
    - An organization may provide multiple services, but a user just needs a single Kerberos password to use all services
  - More convenient where there are many users
  - Requires a Key Distribution Center (KDC)
    - Each service and each user must register a Kerberos principal in the KDC
- 

Kerberos is a very popular framework for company-wide authentication (and authorization, but that will be discussed in a subsequent slide).

It is beyond the scope of this course to describe how to configure the components of a Kerberos deployment. The course will describe the necessary Kafka configurations and assumes the presence of a functional Kerberos environment. For a short tutorial on configuring Kerberos and authenticating to a Kafka cluster over SASL GSSAPI, see <https://qiita.com/visualskyrim/items/8f48ff107232f0befa5a>.

tejaswin.renugunta@walmartlabs.com

# How Kerberos Works



1. Services authenticate with the Key Distribution Center on startup
  - a. Client authenticates with the Authentication Server on startup
  - b. Client obtains a service ticket from Ticket Granting Server
2. Client authenticates with the service using the service ticket

This is very high-level version of how Kerberos works. Refer students to their local security administrator for more information.

# Preparing Kerberos

- Create principals in the KDC for:
    - Each Kafka Broker
    - Application clients
  - Create Keytabs for each principal
    - Keytab includes the principal and encrypted Kerberos password
    - Allows authentication without typing a password
- 

Kafka administrators will have to work with the Kerberos team to create principals (users) for the Brokers and the clients. The Broker principals are often referred to as the "service" principals.

	<p>While keytabs are useful for automation (no password typing) and storing encrypted credentials, the files themselves can be used by attackers to gain access. For this reason, it is important that keytab files be given restricted file permissions.</p>
--	---

# Configuring Kerberos (Broker)

Broker configuration file:

```
listeners = SASL_PLAINTEXT://<host>:<port>    # or SASL_SSL://<host>:<port>
inter.broker.listener.name=SASL_PLAINTEXT      # or SASL_SSL
sasl.kerberos.service.name=kafka

listener.name.sasl_plaintext.gssapi.sasl.jaas.config= \
  com.sun.security.auth.module.Krb5LoginModule required \
  useKeyTab=true \
  storeKey=true \
  keyTab="/etc/security/keytabs/kafka_server.keytab" \
  principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
```

tejaswin.renugunta@walgreens.com

# Configuring Kerberos (Client)

Client configuration file:

```
...
security.protocol=SASL_PLAINTEXT      # or SASL_SSL
sasl.kerberos.service.name=kafka

sasl.jaas.config= \
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
...
...
```

There is no need to configure `sasl.enabled.mechanisms` for Kerberos because GSSAPI is enabled by default.

Clients (Producers, Consumers, Connect workers, etc) will authenticate to the cluster with their own principal (usually with the same name as the user running the client), so obtain or create these principals as needed. Configure listeners and the service name. Adjust the port numbers in the client's `bootstrap.servers` to match the port configured for GSSAPI on the Broker.

- Discuss: what are the security implications of using SASL GSSAPI (Kerberos) with plaintext data transport?
  - Kerberos handles all authentication, and credentials are never passed over the wire, so it is safe to use plaintext data transfer. However, the content of the messages themselves might be sensitive. In this case, it may be advisable to push encryption/decryption onto the clients as discussed earlier. This has its own drawbacks. For example, ksqlDB has no way to programmatically decrypt messages from an event stream like a custom written Consumer would.

# The Kerberos Principal Name

- Kerberos principal
    - Primary [/Instance]@REALM
    - Examples:
      - kafka/kafka1.hostname.com@EXAMPLE.COM
      - kafka-client-1@EXAMPLE.COM
  - Primary is extracted as the default principal name
  - Can customize the username through `sasl.kerberos.principal.to.local.rules`
- 

Like the `principal.builder.class` used with SSL,  
`sasl.kerberos.principal.to.local.rules` is set in the `server.properties` file on the  
brokers. The use of this setting is described in the documentation  
<https://docs.confluent.io/5.2.0/installation/configuration/broker-configs.html>.

tejaswin.renugunta@walgreen.com

# Advanced Security Configurations

- Configure multiple SASL mechanisms on the Broker
  - Useful for mixing internal (e.g., GSSAPI) and external (e.g., SCRAM) clients

```
sasl.enabled.mechanisms = GSSAPI,SCRAM-SHA-256
```

- Configure different listeners for different sources of traffic
  - Useful to designate one interface for clients and one interface for replication traffic:

```
listeners = CLIENTS://kafka-1a:9092,REPLICATION://kafka-1b:9093
```

- Listeners can have any name as long as `listener.security.protocol.map` is defined to map each name to a security protocol:

```
listener.security.protocol.map=CLIENTS:SASL_SSL,REPLICATION:SASL_PLAINTEXT
```
- Specify listener for communication between Brokers and Controller with `control.plane.listener.name`

Use case for mixing GSSAPI (over plaintext) and SCRAM (over SSL): you may want to give limited access to a company's Kafka Cluster to an external partner, but you can't create a Kerberos user (and thus cannot leverage GSSAPI) for the partner since the partner is not an internal employee. Solution: you can open up both SASL SCRAM and SASL Kerberos, letting the partner use the former and the internal employees use the latter.

For multi-homed Brokers (i.e., Brokers with multiple NIC interfaces), you can use the `listener.security.protocol.map` setting to assign specific types of traffic to individual network cards. This is especially useful when isolating ZooKeeper-Broker traffic from clients. For more information, see <https://cwiki.apache.org/confluence/display/KAFKA/KIP-103%3A+Separation+of+Internal+and+External+traffic>.



The `control.plane.listener.name` configuration was added in Apache Kafka 2.2.0.

# Troubleshooting SASL

- To troubleshoot SASL issues
  - Verify all Broker security configurations
  - Verify client security configuration
  - Verify JAAS files and passwords
  - Enable SASL debugging for Kerberos using the `KAFKA_OPTS` environment variable
    - If you are debugging on the Broker, you will have to restart the Broker

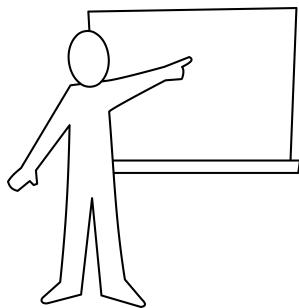
```
$ export KAFKA_OPTS="-Dsun.security.krb5.debug=true"
```

---

SASL configurations the Kafka side are straightforward. Typically, errors in a Kafka environment are due to typos in the username or password in the configuration files.

tejaswin.renugunta@walgreens.com

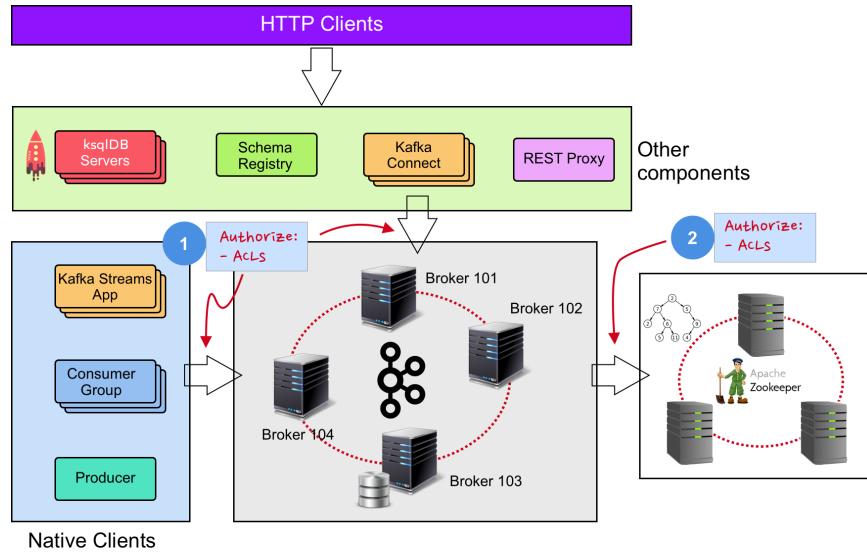
# Module Map



- Overview
- SSL for Encryption and Authentication
- SASL for Authentication
- Authorization ... ←
- Securing the Whole Environment
- Migration to a Secure Cluster
- Security - Confluent Cloud
-  Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# Security - Authorization



1. All relevant resources on the Kafka cluster, such as Topics can be protected by Access Control Lists (ACLs). All clients accessing the Kafka cluster need the corresponding rights to execute `create`, `read` and/or `write` operations on those resources.



Impersonation is not possible without Role Based Access Control (RBAC), that is, the identity of a client accessing the ksqlDB API is not passed through to the Kafka cluster, but rather the ksqlDB Server identifies itself (with its technical account) to the Kafka cluster. With RBAC, the ksqlDB server can impersonate the client in order to give the client access only to what it is authorized to access. RBAC was introduced in CP 5.3 and is not discussed in these materials.

2. All nodes containing meta data about the Kafka cluster, stored in ZooKeeper, can be protected by ACLs. The Brokers need the corresponding rights to read and/or write from and to those nodes

# Authorization Basics

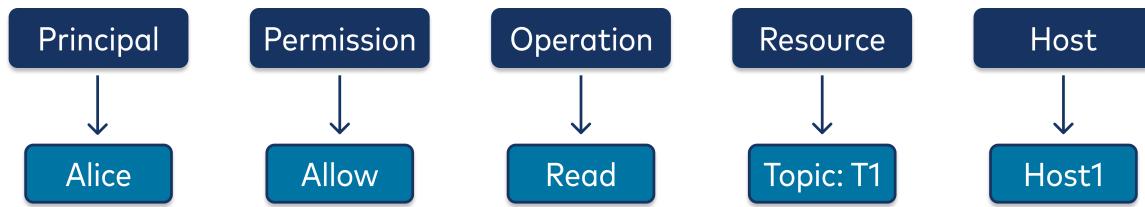
- Authorization controls what permissions each authenticated principal has
  - Authorization is pluggable, with a default implementation
- 

Once we have identified the users accessing the Brokers, the next step is to determine what we will allow them to do in the cluster.

tejaswin.renugunta@walgreens.com

# Access Control Lists (ACLs)

- Example: Alice is allowed to read data from Topic T1 from host Host1



The default authorization plugin implements permissions based on Access Control Lists (ACLs). Authorization is based on a 5-tuple match. We will examine each part individually.

tejaswin.renugunta@walgreens.com

# Principal

- Type + name
  - Supported types: User
    - `User:Alice`
  - Extensible, so users can add their own types (e.g., group)
- 

The default authorizer plugin only allows the use of the `User` type. For other types, administrators will have to install a different authorizer (e.g., Confluent LDAP Authorizer) on all of the Brokers.

tejaswin.renugunta@walgreens.com

# Permissions

- Allow and Deny
    - Deny takes precedence
    - Deny makes it easy to specify “everything but”
  - By default, anyone without an explicit Allow ACL is denied
- 

Once the first ACL is created, the cluster will deny all access that is not explicitly allowed by the ACLs. This is standard security practice since it is easy to identify users who are accidentally locked out of their objects; users granted too much access rarely report the misconfigurations.

tejaswin.renugunta@walgreens.com

# Operations and Resources

- Operations:
  - `Read, Write, Create, Describe, ClusterAction, All`
- Resources:
  - Topic, Cluster, and ConsumerGroup

Operations	Resources
<code>Read, Write, Describe</code> <code>Read</code> and <code>Write</code> imply <code>Describe</code>	Topic
<code>Read</code>	ConsumerGroup
<code>Create, ClusterAction</code> communication between Controller and Brokers	Cluster

For a user to auto-create a Topic if `auto.create.topics.enable` is true, the user will need to issue `Create` as a Cluster operation.

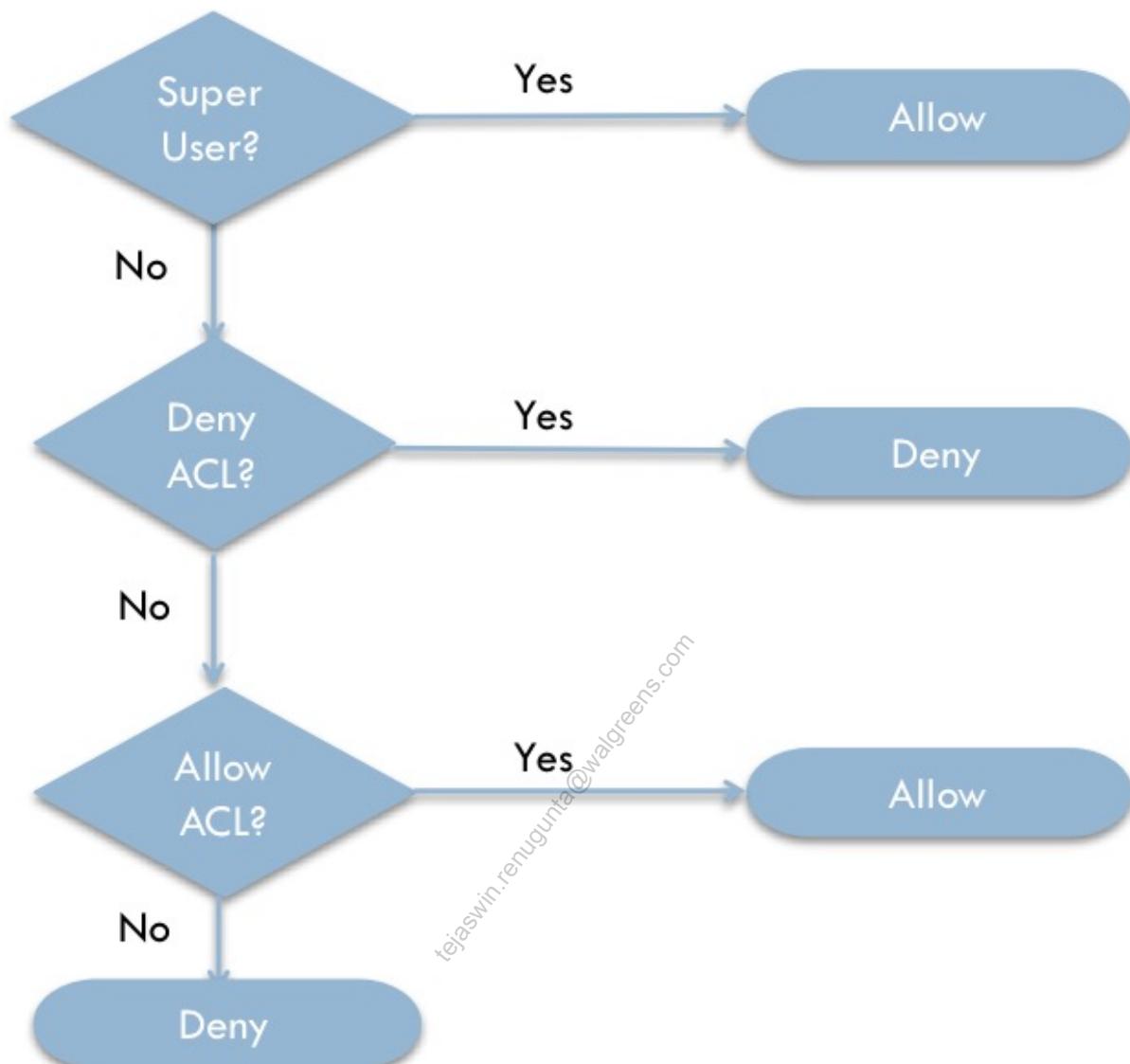
tejaswin.renugunta@walgreens.com

# Hosts

- Allows firewall-type security, even in a non-secure environment
  - Without needing system/network administrators to get involved

tejaswin.renugunta@walgreens.com

# Permission Check Sequence



# Configuring a Broker ACL

- `SimpleAclAuthorizer` is the default authorizer implementation

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
```

- Provides a CLI for adding and removing ACLs
- ACLs are stored in ZooKeeper and propagated to Brokers asynchronously
- ACLs are cached in the Broker for better performance
- Make Kafka principal super users
  - Or grant `ClusterAction` and `Read` on all Topics to the Kafka principal

---

If a different authorizer is needed, install the plugin on the Broker and configure the `authorizer.class.name` to use the new authorizer.

If students need LDAP integration to provide group permissions, they can use the `LdapAuthorizer` from Confluent Enterprise. Modify the `server.properties` file on the Brokers to use the setting `authorizer.class.name=io.confluent.kafka.security.ldap.authorizer.LdapAuthorizer`. Complete instructions on implementing this change can be found at <https://docs.confluent.io/5.0.0/confluent-security-plugins/docs/kafka/introduction.html>

ACLs are stored in ZooKeeper and cached on each Broker periodically so that lookups can be done without contacting ZooKeeper every time.



Active Directory (AD) can integrate with Kerberos for authentication and with Confluent Enterprise LDAP Authorizer Plugin for authorization. This means that all authentication and authorization is handled by a single infrastructure (AD). This is a common use case.

# Configuring ACLs - Producers

- `kafka-acls` can be used to add authorization
- Producer:
  - Grant `Write` on the Topic, `Create` on the Cluster (for Topic auto-creation)
  - Or use `--producer` option in the CLI

```
$ kafka-acls \
  --bootstrap-server kafka-1:9092 \
  --add \
  --allow-principal User:Bob \
  --producer \
  --topic my_topic
```

The `Create` operation on the Cluster (versus on the Topic) is not a typo. For a user to auto-create a Topic if `auto.create.topics.enable` is true, need to issue `Create` as a Cluster action. If you try to add a `Create` operation to a Topic, you will get an error message  `ResourceType Topic only supports operations Read,Write,Describe,All.`

# Configuring ACLs - Consumers

- Consumer:
  - Grant **Read** on the Topic, **Read** on the ConsumerGroup
  - Or use the **--consumer** option in the CLI

```
$ kafka-acls \
  --bootstrap-server kafka-1:9092 \
  --add \
  --allow-principal User:Bob \
  --consumer \
  --topic my_topic \
  --group group1
```

tejaswin.renugunta@walgreens.com

# Removing Authorization

- `kafka-acls` can be used to remove or change authorization
  - May use additional options, e.g. `deny-principal`, `remove`, etc
- If needed, also useful to revoke authorization after connections are established
  - SSL and SASL authentication happens only once during the connection initialization process
  - Since no re-authentication occurs after connections are established:
    - Use `kafka-acls` to remove all permissions for a principal
    - All requests on that connection will be rejected
    - Reset connections as needed

---

Whether using SSL or SASL, performance would be terrible if every connection were authenticated. However, leaving the connection open means that there has to be a way to disable a client's access if we cannot end the connection immediately. The most immediate way to affect a client is to remove their authorization using ACLs.

# Wildcard Support (1)

- Allow user **Jane** to produce to any Topic whose name starts with "Test-"

```
$ kafka-acls --bootstrap-server kafka-1:9092 \
--add \
--allow-principal User:Jane \
--producer --topic Test- \
--resource-pattern-type prefixed
```

- Allow all users **except BadBob** and all hosts **except 198.51.100.3** to read from **Test-topic**:

```
$ kafka-acls --bootstrap-server kafka-1:9092 --add \
--allow-principal User:'*' \
--allow-host '*' \
--deny-principal User:BadBob \
--deny-host 198.51.100.3 \
--operation Read --topic Test-topic
```

The first example shows that we can pattern match prefixes by using **--resource-pattern-type prefixed**.

The second example shows that you can allow access to a resource for everyone except certain specific hosts or principals.

# Wildcard Support (2)

- List all ACLs for the topic `Test-topic`:

```
$ kafka-acls --bootstrap-server kafka-1:9092,kafka-2:9092 \
  --list --topic Test-topic \
  --resource-pattern-type match
```

In this example, notice `--resource-pattern-type match`. This matches any ACLs created with literal, wildcard ('\*'), or prefixed resource patterns ('Test-', like the first example). If we don't use this option, it will only match ACLs that were created that literally spell out `Test-topic` rather than using a prefix or wildcard.

It is possible to make an ACL for everyone in the cluster using the `--cluster` option. Remember that you can make "allow" ACLs and "deny" ACLs that affect the same entity, and the permission check sequence will ensure the "deny" rule is checked before any "allow" rules.

Prefixed wildcard support (the use of a string before the \* character) was added in Kafka 2.0.0. See <https://docs.confluent.io/current/kafka/authorization.html#adding-acls> for more information.

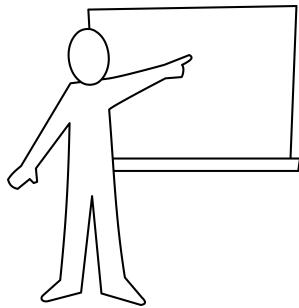
# Troubleshooting Kafka Authorization

- Verify all Broker security configurations
- Verify client security configuration
- Enable **DEBUG** level in Kafka authorizer in `/etc/kafka/log4j.properties`
  - Logs the decision on every request
  - Can serve as an audit log

```
log4j.logger.kafka.authorizer.logger=DEBUG, authorizerAppender
```

tejaswin.renugunta@walgreens.com

# Module Map



- Overview
- SSL for Encryption and Authentication
- SASL for Authentication
- Authorization
- Securing the Whole Environment ... ←
- Migration to a Secure Cluster
- Security - Confluent Cloud
-  Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# ZooKeeper Security is Important

- ZooKeeper stores:
    - Critical Kafka metadata
    - ACLs
  - We need to prevent untrusted users from modifying this data
- 

As the source of truth for cluster configurations, ZooKeeper should be secured so that unauthorized users cannot corrupt the cluster or change their own access rights.

tejaswin.renugunta@walgreens.com

# ZooKeeper Security Integration

- Set the authentication provider in `zookeeper.properties`

```
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

- Create a server JAAS configuration file, here called `zookeeper_jaas.conf`:

```
Server {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    keyTab="/path/to/server/keytab"  
    storeKey=true  
    useTicketCache=false  
    principal="zookeeper/zk.hostname.com@EXAMPLE.COM";  
};
```

- Add the JAAS file to the JVM:

```
$ export KAFKA_OPTS="-  
Djava.security.auth.login.config=/etc/kafka/zookeeper_jaas.conf"  
$ zookeeper-server-start /etc/kafka/zookeeper.properties
```

- Each ZooKeeper path should be writable by its creator, readable by all

ZooKeeper supports authentication through SASL (Kerberos or Digest M5). For more information, see

[https://docs.confluent.io/current/tutorials/security\\_tutorial.html#configure-zk](https://docs.confluent.io/current/tutorials/security_tutorial.html#configure-zk).

For more ZooKeeper-specific information, see

<https://cwiki.apache.org/confluence/display/ZOOKEEPER/Client-Server+mutual+authentication>.



SSL is not supported in ZooKeeper version 3.4.

# Broker as ZooKeeper Client

- Set `zookeeper.set.acl` to `true` on Broker
- Configure the Broker as a ZooKeeper client with JAAS file, here called `zkclient_jaas.conf`:

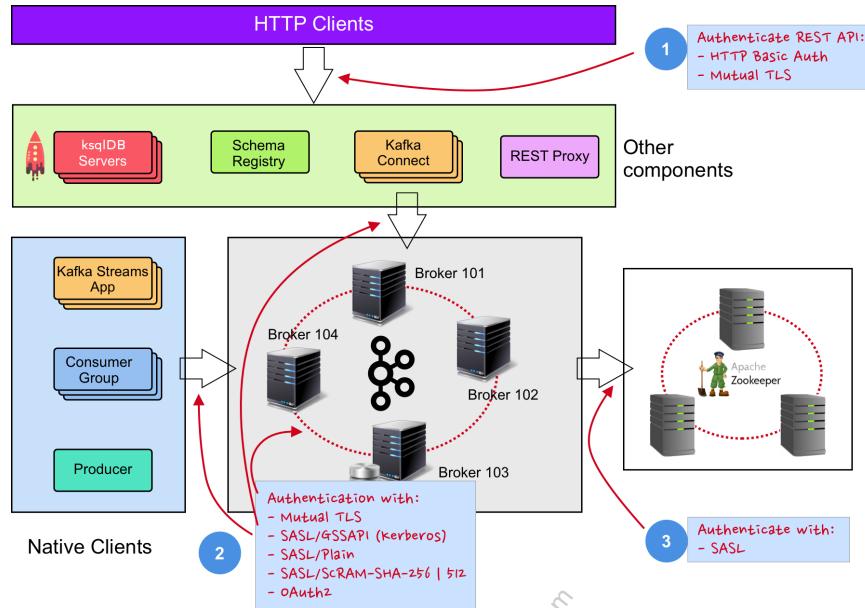
```
Client {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_server.keytab"  
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";  
};
```

- Add the JAAS file to the JVM:

```
$ export KAFKA_OPTS="-  
Djava.security.auth.login.config=/etc/kafka/zkclient_jaas.conf"  
$ kafka-server-start /etc/kafka/zookeeper.properties
```

Note that this file is the JAAS file on a Broker, since the Broker is the client of ZooKeeper.

# Securing Schema Registry and REST Proxy



This slide reminds us of the various connections that need to be secured. The next two slides focus on Schema Registry and REST Proxy since they have specific Confluent Enterprise security plugins that make it possible to **authorize** REST Proxy and Schema Registry REST API clients.

# Securing The Schema Registry

- Secure communication between REST client and Schema Registry (HTTPS):
  - HTTP Basic Authentication
  - SSL (transport)
- Secure transport and authentication between the Schema Registry and the Kafka cluster:
  - SSL (transport)
  - SASL (authentication)
  - Mutual SSL (transport + authentication)
- Confluent Enterprise **security plugin**:
  - Restricts schema evolution to administrative users
  - Client application users get read-only access

---

## Resources:

- HTTP Basic AuthN for all REST API services:  
<https://docs.confluent.io/current/security/basic-auth.html>
- Schema Registry HTTPS: <https://docs.confluent.io/current/schema-registry/>
- Schema registry security plugin: <https://docs.confluent.io/current/confluent-security-plugins/schema-registry/introduction.html>
- Schema Registry configuration reference: <http://docs.confluent.io/current/schema-registry/docs/config.html>

## Version notes:

- The Schema Registry security plugin was introduced in CP 4.0
- As of CP 5.0, Schema Registry can use Kafka itself to facilitate its leader elections, and thus does not need direct access to ZooKeeper.

# Securing The REST Proxy

- Secure communication between REST clients and the REST Proxy (HTTPS)
  - HTTP Basic Authentication
  - SSL (transport)
- Secure communication between the REST Proxy and Apache Kafka
  - SSL (transport)
  - SASL (authentication)
  - Mutual SSL (transport + authentication)
- Confluent Enterprise **security plugin**:
  - Propagates client principal authentication to Kafka Brokers
  - More granular than single authentication for all clients

---

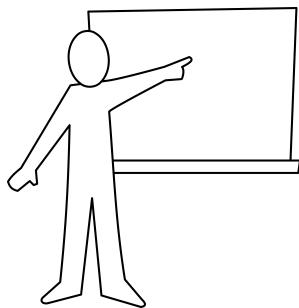
## Resources:

- HTTP Basic AuthN for all REST API services:  
<https://docs.confluent.io/current/security/basic-auth.html>
- REST Proxy HTTPS: <https://docs.confluent.io/current/kafka/encryption.html#rest-proxy>
- REST Proxy security plugin: <https://docs.confluent.io/current/confluent-security-plugins/kafka-rest/introduction.html>
- REST Proxy configuration reference: <http://docs.confluent.io/current/kafka-rest/docs/config.html>

## Version notes:

- Since Confluent Platform 4.0: REST proxy security plugin propagates client principal authentication to Kafka Brokers

# Module Map



- Overview
- SSL for Encryption and Authentication
- SASL for Authentication
- Authorization
- Securing the Whole Environment
- Migration to a Secure Cluster ... ←
- Security - Confluent Cloud
-  Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# Migrating Non-Secure to Secure Kafka Cluster

1. Configure Brokers with multiple ports

```
listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

2. Gradually migrate clients to the secure port
3. When done, turn off **PLAINTEXT** listener on all Brokers

---

This procedure assumes that the cluster's mission critical Topics are replicated so that a rolling reboot can be performed to implement the changes to the **server.properties** file.

tejaswin.renugunta@walgreens.com

# Migrating from a Non-Secure to a Secure ZooKeeper

On each Broker:

1. Create ZooKeeper client JAAS configuration file and add it to the JVM (see earlier slide)
2. Set `zookeeper.set.acl` property to `false`
3. Restart
4. Set `zookeeper.set.acl = true`
5. Restart again
6. Use the `zookeeper-security-migration` tool to set permissions on existing ZooKeeper paths:

```
$ zookeeper-security-migration \
--zookeeper.acl=secure \
--zookeeper.connect=zk_host:2181
```

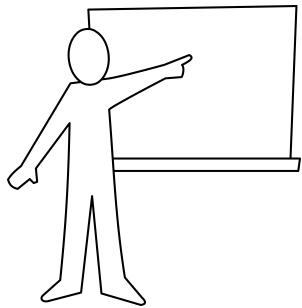
---

The reason Brokers need to be restarted twice:

1. First restart: enable security in every Broker but create znodes without ACLs
2. Second restart: enable creation of znodes with ACLs. All Brokers will have ZK security enabled at that point

If you were to set `zookeeper.set.acl true` in the first restart, then Brokers that hadn't restarted yet won't be able to access znodes created by the Brokers that have restarted.

# Module Map

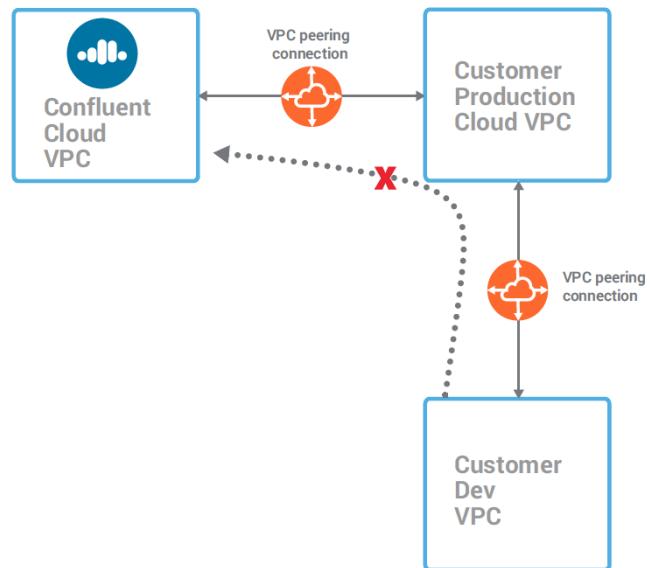


- Overview
- SSL for Encryption and Authentication
- SASL for Authentication
- Authorization
- Securing the Whole Environment
- Migration to a Secure Cluster
- Security - Confluent Cloud ... ←
- 🔑 Hands-on Lab: **Securing the Kafka Cluster**

tejaswin.renugunta@walgreens.com

# Security - Confluent Cloud

- Security enabled **Out-of-the-Box**
- Storage isolation
- VPC peering possible
- User → Cloud communication secured by TLS
- Data encrypted in motion & at rest
- CCloud is hosted in multiple AWS, GCP, and Azure regions



- Confluent Cloud has all security features enabled out-of-the-box, with no extra effort and at no extra cost, and provides full SOC-2 and PCI Level-1 compliance (HIPAA coming soon!).
- Confluent Cloud Enterprise customers also have storage isolation and they can choose to create VPC peering connection between Confluent Cloud and their own VPCs for an extra layer of security.
- Customers access Confluent Cloud via three main interfaces:
  - Confluent Cloud web-based user interface
  - Confluent Cloud command line client
  - Apache Kafka protocol

All communication between users and Confluent Cloud is secured using TLS encryption.

- **Encryption:** Confluent Cloud encrypts all data in motion and at rest.
- **Physical security:** Confluent Cloud is hosted in multiple AWS, GCP, and Azure regions

# Hands-On Lab

- In this Hands-On Exercise, you will configure one-way SSL authentication, two-way SSL authentication, and explore the SSL performance impact
- Please refer to **Lab 06 Kafka Security** in the Exercise Book:
  - a. **Securing the Kafka Cluster** in Exercise Book



tejaswin.renugunta@walgreens.com

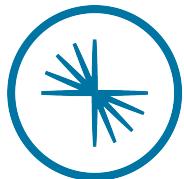
# Module Review



- Kafka supports both authentication and authorization
  - Authentication via SSL or SASL
  - Authorization via ACLs (or other pluggable systems)
- SSL provides encryption of data in flight, if required
- Authorization and authentication do not require code changes
  - Just changes to configuration files

tejaswin.renugunta@walgreens.com

# 07 Data Pipelines with Kafka Connect



CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Providing Durability
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance
6. Kafka Security
7. Data Pipelines with Kafka Connect ... ←
8. Kafka in Production
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

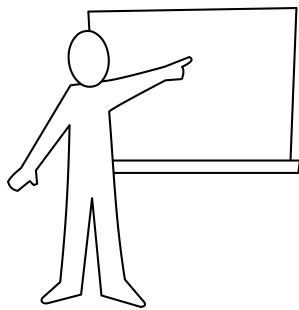


After this module you will be able to:

- explain the motivation for Kafka Connect
- list commonly used Connectors
- explain the differences between standalone and distributed mode
- configure and use Kafka Connect
- use Single Message Transforms (SMTs)

tejaswin.renugunta@walgreens.com

# Module Map



- The Motivation for Kafka Connect ... ←
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
-  **Hands-on Lab: Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# GOALS



1. Focus on copying
2. Batteries included
3. Standardize
4. Parallelism
5. Scale

---

The goals of a framework that provides integration with Kafka are:

1. **Focus on copying:** It is all about getting data in and getting data out. Nothing more
2. **Batteries included:** The Kafka platform should include this framework and includes the most popular plugins for import and export of data
3. **Standardize:** The framework should standardize the task of importing and exporting data. No one-off solutions!
4. **Parallelism:** The framework should be able to parallelize the work when possible (e.g., importing a set of tables or files).
5. **Scale:** We want to be able to scale out the workload, similar to how we can scale out brokers or consumer groups

# What is Kafka Connect?

- Kafka Connect is a framework for streaming data between Apache Kafka and other data systems
- Kafka Connect is open source, and is part of the Apache Kafka distribution
- It is simple, scalable, and reliable



Kafka Connect is not an API like the Client API (which implements Producers and Consumers within the applications you write) or Kafka Streams. It is a reusable framework that uses plugins called Connectors to customize its behavior for the endpoints you choose to work with.

tejaswin.renugunta@walgreens.com

# Connect Basics

- **Connectors** are logical jobs that copy data between Kafka and another system
- **Source** Connectors read data *from* an external data system into Kafka
  - Internally, a source connector is a Kafka Producer
- **Sink** Connectors write Kafka data *to* an external data system
  - Internally, a sink connector is a Kafka Consumer Group

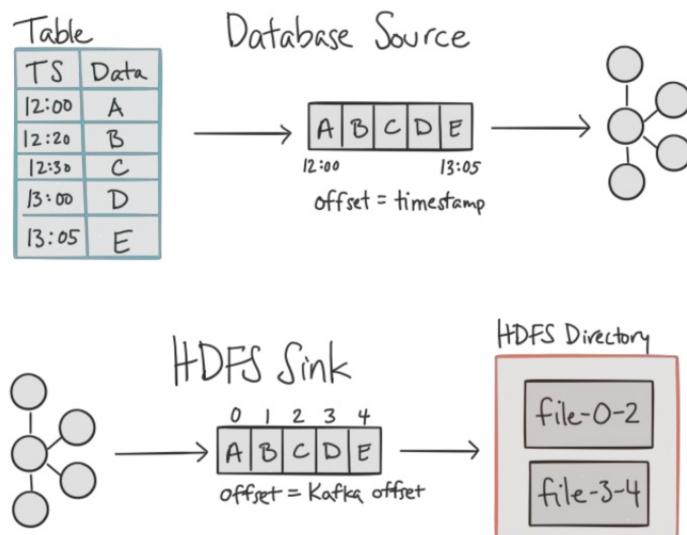
---

An instance of Connect (called a worker) can function as both a producer and consumer, depending on the type of connector you install. In fact, internally connectors are running the standard client code: Producer (source connectors) and Consumer (sink connectors).

Source connectors read from external data stores (e.g., databases). Sink Connectors pull data from a Kafka topic and write it out to an external application (e.g., HDFS, Elasticsearch).

The term "Connector" can be used for either the type of plugin (e.g., the HDFS Connector) or a configured instance of a connector (e.g., the mycompany\_HDFS connector).

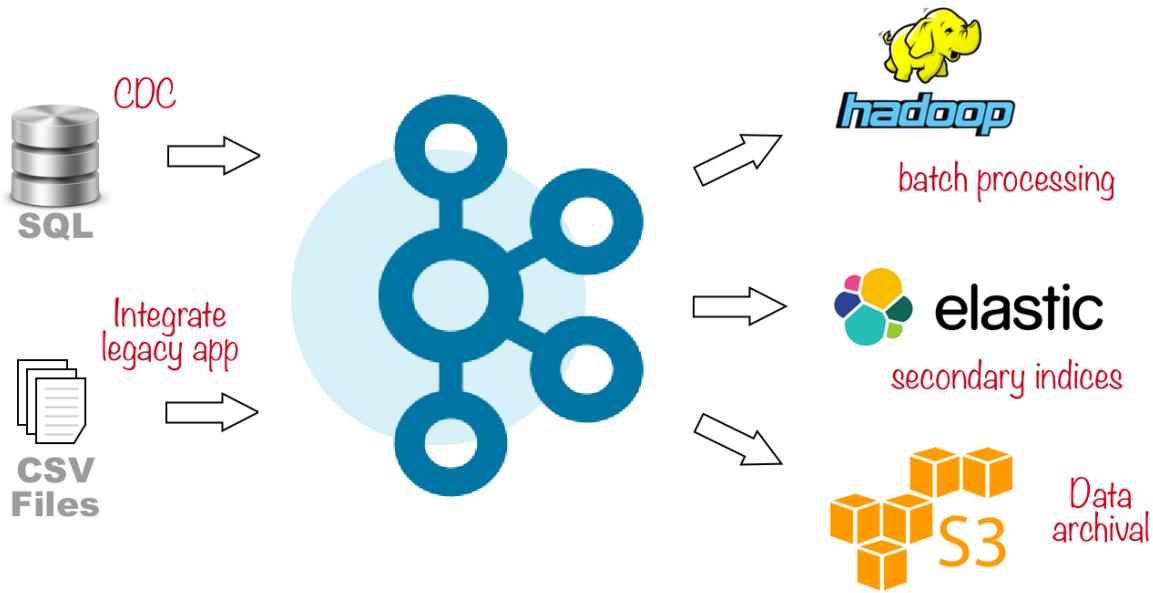
# Connect Illustrated



Here we see a sample source and a sample sink.

- **Source:** Data originating in a relational DB is imported into Kafka. The timestamp from the table is "translated" into what's the offset in Kafka
- **Sink:** Data from Kafka is exported to a HDFS directory. The Kafka offset becomes part of the filenames in the sink (i.e. from → to)

# Example Use Cases



- Example use cases for Kafka Connect include:
  - Stream an entire SQL database into Kafka
    - Bulk - load entire table
    - Change data capture (CDC) - load table changes as they happen
  - Import CSV files generated by legacy app into Kafka
  - Stream Kafka Topics into Hadoop File System (HDFS) for batch processing
  - Stream Kafka Topics into Elasticsearch for secondary indexing
  - Archive older data in low cost object storage
    - e.g. Amazon Simple Storage Service (S3)

Swetha Nugunta@walgreens.com

# Why Not Just Use Producers and Consumers?



## Not Optimal

- One-off tools
- all-purpose tools
- Stream processing frameworks



## Kafka Connect

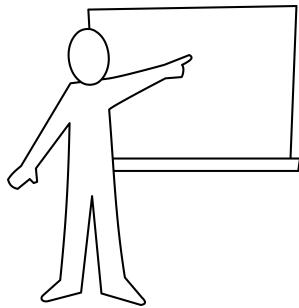
- Off-the-shelf
- Tested
- Connectors for common sources & sinks
- Fault tolerance
- Automatic load balancing
- No coding required
- Pluggable/extendable by developers

Typically the Client API is included in your application as a library. But what if you don't have code access? You'll need to write a client which can behave as a producer or consumer between the external applications and the cluster. Writing your own producers and consumers provides complete flexibility to send any data to Kafka or process it in any way, but this flexibility means you do everything yourself.

Connect, on the other hand, is a framework that is purpose-built for getting data in and out of Kafka. This framework is written to make Connect highly available and scalable. Kafka Connect relies on plugins called **Connectors** to move data between Kafka specific sources and sinks. The best part about connectors is that many are already available for the most common sources and sinks, and many are open source and free to use. If a connector is not available, it is possible to create your own custom connector and receive all the benefits of the Kafka Connect framework.

Connect is not designed to do heavy transformations like Kafka Streams, although it does support light transformations in the form of Single Message Transforms (SMTs). We will discuss SMTs in more detail later in this module, and we will compare and contrast SMTs vs. Kafka Streams vs. ksqlDB in an upcoming module.

# Module Map



- The Motivation for Kafka Connect
- Types of Connectors ... ↵
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
-  Hands-on Lab: **Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# Confluent Hub

CONFLUENT

Product Cloud Developers Blog Docs DOWNLOAD

## Confluent Hub

### Discover Kafka® connectors and more

What plugin are you looking for?

#### Filters

Plugin type

- Sink
- Source
- Transform
- Converter

Enterprise support

- Confluent supported
- Partner supported
- None

Verification

- Confluent built
- Confluent Tested
- Verified gold
- Verified standard
- None

License

- Commercial
- Free

#### Results (158)

+ Submit a plugin

**Kafka Connect GCP Pub-Sub**

SOURCE CONNECTOR

A Kafka Connect plugin for GCP Pub-Sub

Available fully-managed on Confluent Cloud

Enterprise support: Confluent supported	Verification: Confluent built	License: Commercial
Installation: Confluent Hub CLI, Download	Author: Confluent, Inc.	Version: 1.0.2



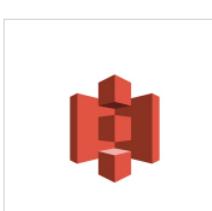
**Kafka Connect S3**

SINK CONNECTOR

The S3 connector, currently available as a sink, allows you to export data from Kafka topics to S3 objects in either Avro or JSON formats

Available fully-managed on Confluent Cloud

Enterprise support: Confluent supported	Verification: Confluent built	License: Free
Installation: Confluent Hub CLI, Download	Author: Confluent, Inc.	Version: 5.5.1



This slide shows Confluent Hub, your source for over 100 Kafka connectors. Using the available filters, you can search for Kafka connectors based upon type, support availability, verification level, license requirement, and availability on Confluent Cloud.

You can find the Confluent Hub on the web at <https://confluent.io/hub>

## Source Connector examples:

- DataGen
  - great for generating test data and doing load testing
  - feed it custom schemas to authentically simulate your use case
- JDBC
  - great for getting SQL database tables into Kafka
- Confluent Replicator
  - great for migrating data from one kafka cluster to another
- IoT Hub
  - get data from Azure IoT Hub into Kafka

## Sink Connector examples:

- Apache Druid
  - streaming analytics database that plays nicely with Kafka
- ElasticSearch
  - index everything
- AWS Simple Storage Service (S3)
  - archive data in S3 at high throughput
- Google Cloud Storage (GCS)
  - archive data in GCS at high throughput

For a detailed description of Confluent Hub:

<https://www.confluent.io/blog/introducing-confluent-hub/>



A goal of Confluent is to provide a comprehensive set of supported connectors.

# Installing New Connectors

## From Confluent Hub

- Use `confluent-hub` client included with Confluent Platform

```
$ confluent-hub install debezium/debezium-connector-mysql:latest
```

## From other sources

- Package Connector in JAR file
- Install JAR file on **all** Kafka Connect worker machines
  - Connectors are installed as plugins
  - There is **library isolation** between plugins

```
plugin.path=/path/to/my/plugins
```

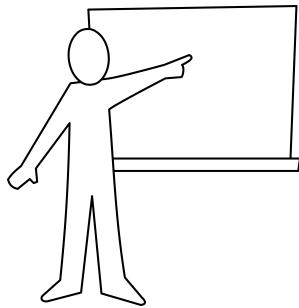
Prior to Kafka 0.11.0, JARs were placed in a path specified by `CLASSPATH` and there was no library isolation between connectors

```
$ export CLASSPATH="$CLASSPATH:/path/to/my/connectors/*"
```

**Question:** Why is library isolation important?

**Answer:** Connectors built by different developers might use different versions of the same library.

# Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation ... ↪
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
-  Hands-on Lab: **Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# Kafka Connect Reliability and Scalability

- Elasticity and scalability
  - To add resources to Connect simply start more workers
  - Connect Workers discover each other and load-balance the work automatically
  - Allows running more connectors and more tasks
- Reliability and fault-tolerance
  - If a worker fails, then all its tasks and connectors will automatically restart on another worker
  - If individual tasks or connectors fail, then the user can decide how to respond
- ALL connectors support at-least-once semantics
- SOME connectors support exactly-once semantics
- MANY connectors support schema evolution
  - Data format changes in source system will reflect in Kafka and all target systems

---

Kafka Connect is fault tolerant to failed workers, but not to individual failed tasks. If failed tasks were automatically restarted, then a "poison pill" message may cause an infinite restart loop. Instead, the user must investigate why the task failed, perhaps take manual steps to remove a problematic record, and then resubmit the individual task using the Kafka Connect REST API.

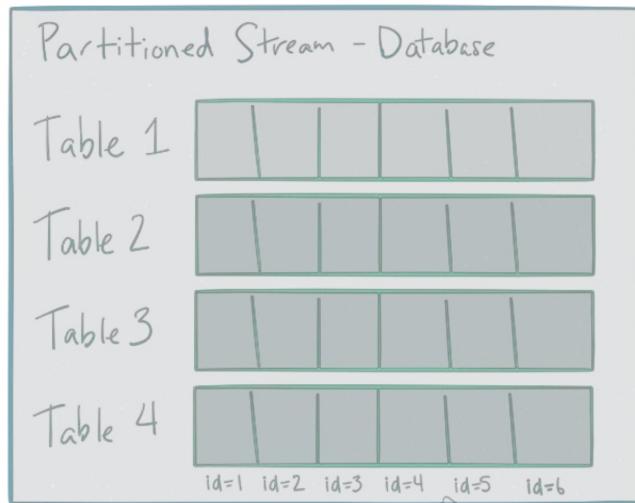
# Providing Parallelism and Scalability (1)

- In Kafka, we split a topic (stream) into partitions to parallelize work.



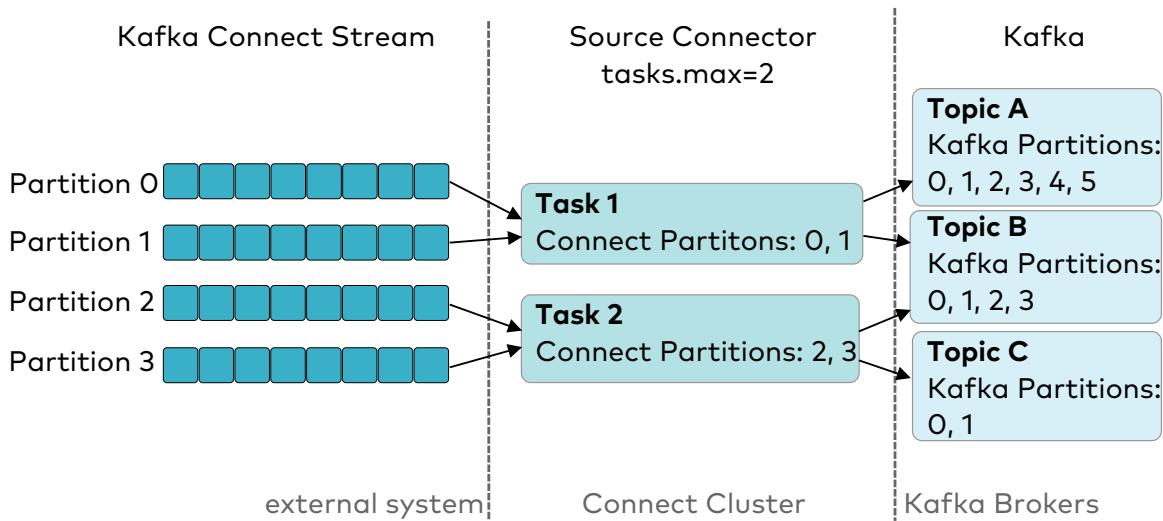
# Providing Parallelism and Scalability (2)

- For a Source Connector, a **partition** is a subset of data in the source system



Kafka Connect also uses the concept of partitioning the input stream of data. Admittedly it is a bit confusing to use the same terminology for two different things. In the case of Connect, the term "Partition" can mean any subset of data in the source or the sink. How a Partition is represented depends on the type of Connector (e.g., tables are the "Partitions" for the JDBC database connector, files are the "Partitions" for the FileStream connector).

# Providing Parallelism and Scalability (3)



- Splitting the workload into smaller pieces provides the parallelism and scalability
- Connector jobs are broken down into *tasks* that do the actual copying of the data
- *Workers* are processes running one or more tasks, each in a different thread

Pictured, we see an external system whose data is imported to Kafka by a source connector. The source connector defines 2 tasks. The "Connect Partitions" are assigned to those tasks. The tasks are the threads that actually move the data. In this case, Task 1 produces data from the external system to topics A and B in Kafka. In parallel, Task 2 produces data to topics B and C. Notice that the number of "Connect Partitions" and the number of "Kafka Partitions" are unrelated. Also notice that the task threads are running in a "connect cluster," not on Kafka brokers. We will discuss deployment in upcoming slides.

Think about this image in terms of a database source connector. The "Connect Partitions" would be tables of the database. Each table would be produced to its own topic in Kafka.

# Providing Parallelism and Scalability (4)

- Partitioning → parallelism & scalability
  - 1 Connector job → 1..n **tasks**
  - 1 Worker → 1..n **tasks**
  - 1 Task per Thread
- 

A Connector is a Connector class and a configuration. Each connector defines and updates a set of tasks that actually copy the data. Connect distributes these tasks across workers.

Worker processes can be deployed like any application process (container in a Kubernetes pod, Ansible, Puppet, Chef, entrypoint script, etc.). Kafka manages coordination and task scheduling via the Kafka Group Coordination Protocol, just like Consumer Group Coordination (discussed more in upcoming slides).



not all connectors support multiple tasks and parallelism. For example, the **syslog** source connector only supports one task.

# Source and Sink Offsets

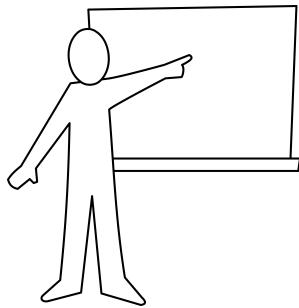
- Kafka Connect tracks the produced and consumed offsets so it can restart at the correct place in case of failure
- What the offset corresponds to depends on the Connector, for example:
  - File input: offset → position in file
  - Database input: offset → timestamp or sequence id
- The method of tracking the offset depends on the specific Connector
  - Each Connector can determine its own way of doing this
- Examples of source and sink offset tracking methods:
  - JDBC source in distributed mode: a special Kafka Topic
  - HDFS sink: an HDFS file
  - FileStream source: a separate local file

---

As with Kafka topics, the position within the partitions used by Connect must be tracked as well to prevent the unexpected replay of data. Just as different connectors use the word "partition" differently, the object used as the "offset" varies from connector to connector, as does the method of tracking the offset. The most common way to track the offset is through the use of a Kafka topic, though the Connector's developer can use whatever they want.

Source connectors tend to vary with how they track offsets. With sink connectors, the offsets are Kafka topic partition offsets, and they're usually stored in the external system where the data is being written.

# Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes ... ↪
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
-  Hands-on Lab: **Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# Two Modes: Standalone and Distributed

## Standalone mode

- 1 worker on 1 machine
- When to use:
  - testing and development
  - non distributable process

## Distributed mode

- n workers on m machines
- When to use:
  - production environments
  - for **fault tolerance**
  - and **scalability**

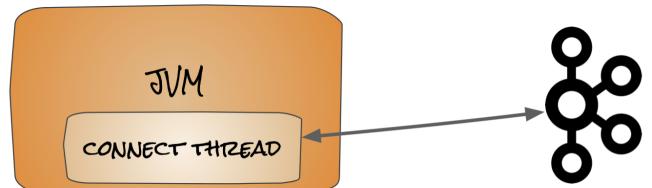
tejaswin.renugunta@walgreens.com

# Running in Standalone Mode

To run in standalone mode, start a process by providing as arguments

- Standalone config properties file
- 1...n connector config files

	Each connector instance runs in own thread
---	--



```
$ connect-standalone connect-standalone.properties \
  connector1.properties [connector2.properties connector3.properties ...]
```

A single worker can run as many connectors as you wish. Once the connectors are running, changes to their configuration files will only take effect on restart of the worker. Making changes to the connectors without a restart will be discussed later in the chapter.

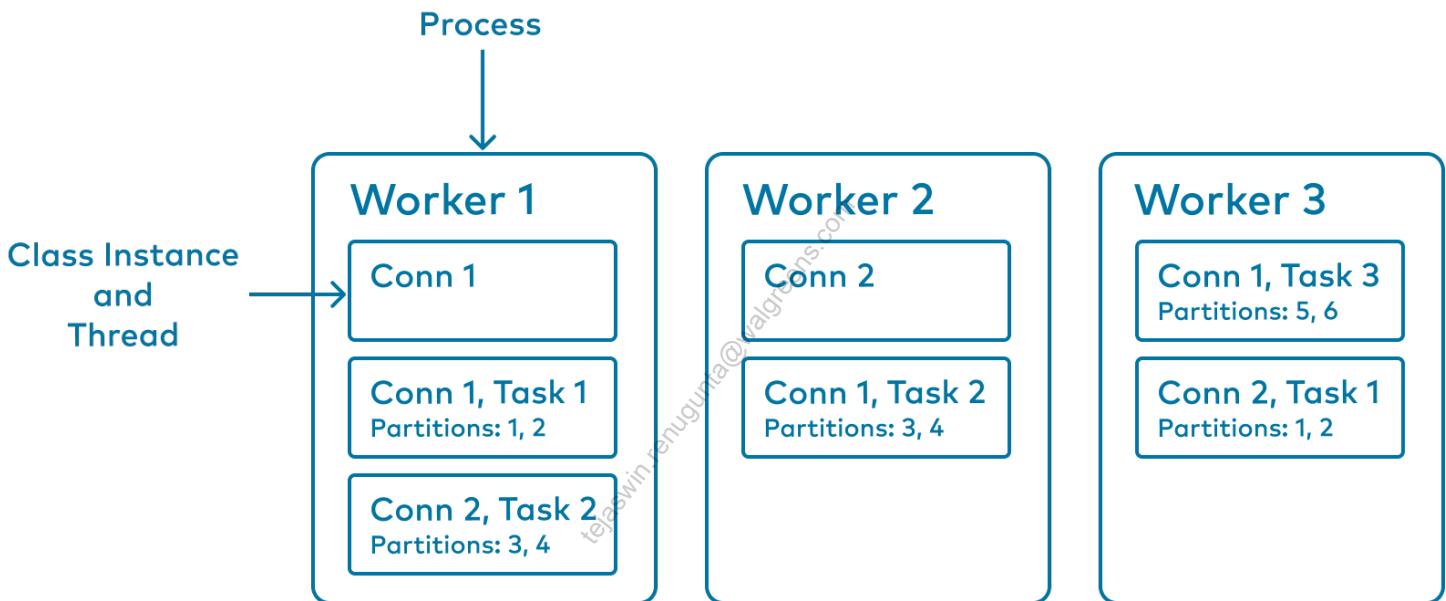
# Running in Distributed Mode

Start Kafka Connect **on each** worker node

```
$ connect-distributed connect-distributed.properties
```

## Group coordination

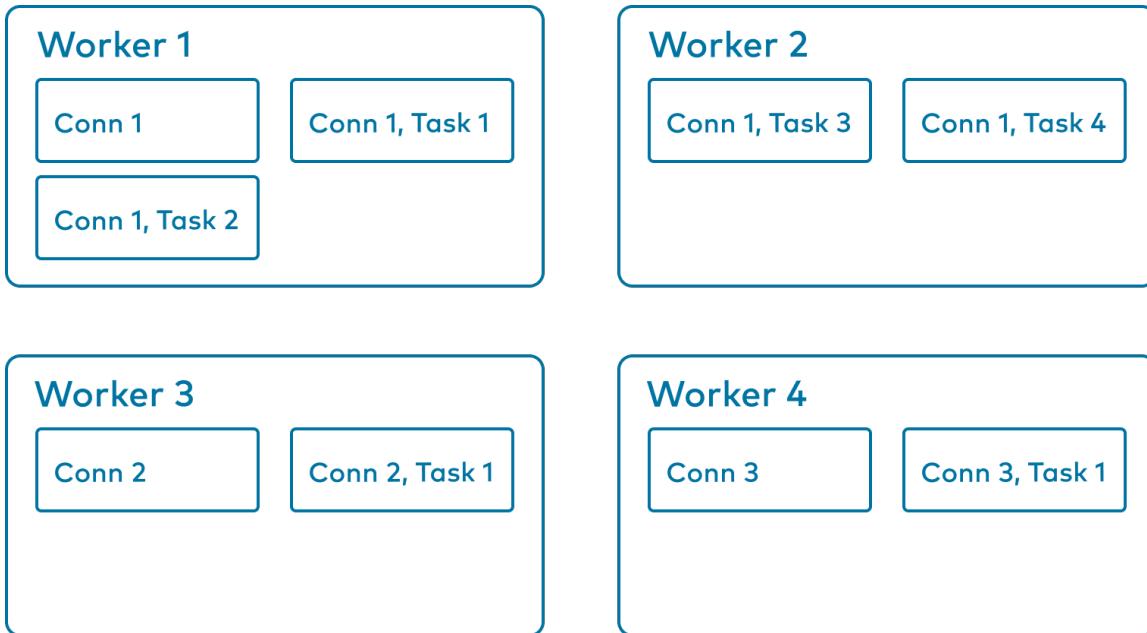
- Connect leverages Kafka's group membership protocol
  - Configure workers with the same `group.id`
- Workers distribute load within this Kafka Connect "cluster"



In distributed mode, the connector configurations **cannot** be kept on the Worker systems; a failure of a worker should not make the configurations unavailable. Instead, distributed workers keep their connector configurations in a special Kafka topic which is specified in the worker configuration file.

Workers coordinate their tasks to distribute the workload using the same mechanisms as Consumer Groups.

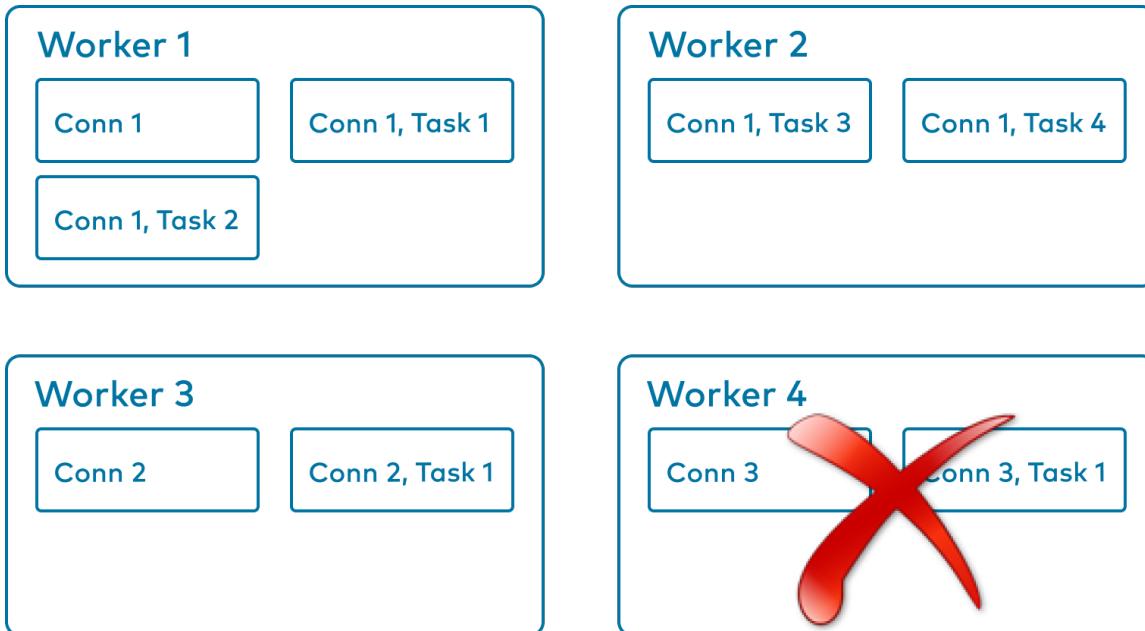
# Fail-over 1/3



Kafka Connect, when run in distributed mode, automatically and transparently does the failover for you. Here we have 4 workers with 3 connectors A, B, C. Connector A has 4 tasks running on workers 1 and 2 whilst connectors B and C each have a single task running on worker 3 and 4 respectively.

tejaswin.renugunta@wazeens.com

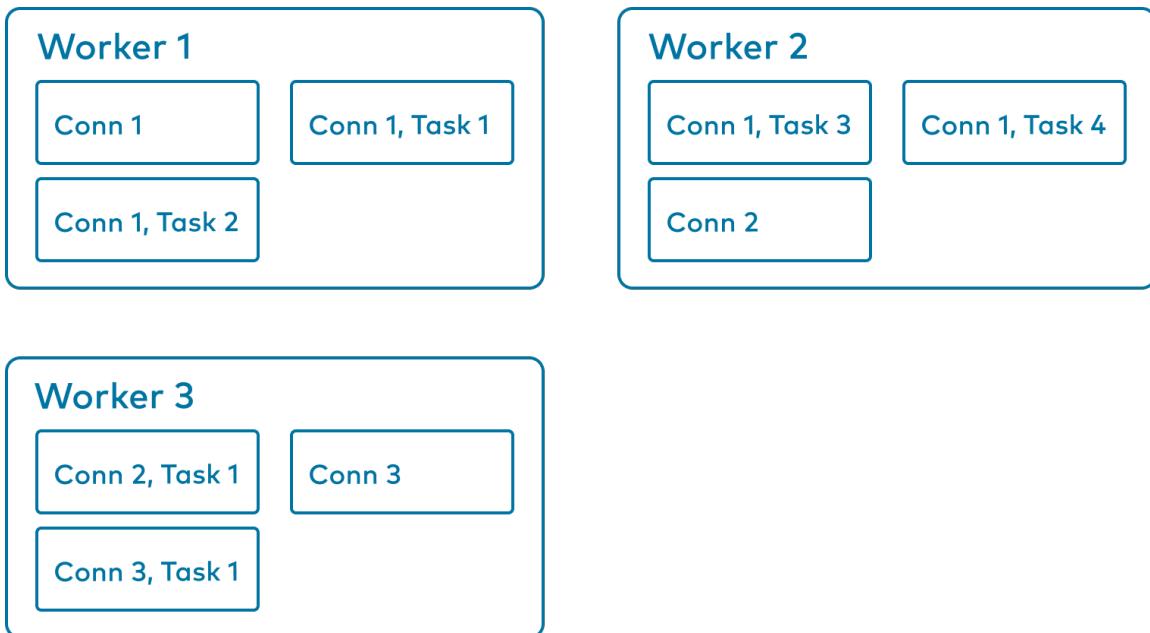
## Fail-over 2/3



Let's now assume that worker 4 fails... what happens to connector C with its task C1?

tejaswin.renugunta@wizbuddhians.com

# Fail-over 3/3



As we can see, Kafka Connect uses a similar technique that we know from consumer groups to reschedule or reallocate workload evenly across the available workers. In our case the connector job B has been moved to worker 2 whilst the connector job C and task C1 have been moved to worker 3.

Rebalancing/task assignment is performed by the Worker, not by the Connector instance thread. Much like consumer groups, there's a concept of a "leader" Worker which performs task reassessments whenever rebalancing is triggered, which can involve movement of connector threads in addition to tasks. See <https://github.com/apache/kafka/blob/2.2/connect/runtime/src/main/java/org/apache/kafka/connect/runtime/distributed/WorkerCoordinator.java#L220-L258> for more detail.

# Configuring Workers: Both Modes

- You can modify Connect configuration settings
  - Distributed mode in `/etc/kafka/connect-distributed.properties`
  - Standalone mode in `/etc/kafka/connect-standalone.properties`
- Important configuration options common to all workers:

Property	Description
<code>bootstrap.servers</code>	A list of host/port pairs to use to establish the initial connection to the Kafka cluster
<code>key.converter</code>	Converter class for the key
<code>value.converter</code>	Converter class for the value

	Customize producer (source) or consumer (sink) properties using prefixes, for example: <code>producer.batch.size=2048</code>
---	--

Because Connect Workers are based on the Client API, some of the same settings (e.g., `bootstrap.servers`) will be used.

Connect Converters are like wrappers around Serializers and Deserializers. Since a Worker can be a producer or a consumer (depending on the Connector), it was more efficient to specify one object rather than two when configuring the Worker.

	See <a href="https://docs.confluent.io/current/connect/references/allconfigs.html">https://docs.confluent.io/current/connect/references/allconfigs.html</a> for a comprehensive list of configuration settings.
---	---

# Configuring Workers: Standalone Mode

Property	Description
<code>offset.storage.file.filename</code>	The filename in which to store offset data for the Connectors (Default: ""). This enables a standalone process to be stopped and then resume where it left off.

The provided file `/etc/kafka/connect-standalone.properties` sets this property to `/tmp/connect.offsets` which is not a very durable location on disk

tejaswin.renugunta@walgreens.com

# Configuring Workers: Distributed Mode

Property	Description
<code>group.id</code>	A unique string that identifies the Kafka Connect cluster group the worker belongs to
<code>session.timeout.ms</code>	Timeout used to detect worker failures
<code>heartbeat.interval.ms</code>	Time between heartbeats to the group coordinator. Must be smaller than <code>session.timeout.ms</code>
<code>config.storage.topic</code>	Topic in which to store Connector & task config. data
<code>offset.storage.topic</code>	Topic in which to store offset data for Connectors
<code>status.storage.topic</code>	Topic in which to store connector & task status
<code>topic.creation.enable</code>	Allow source connector configurations to define topic creation settings (Default: true)

As with Consumer Groups, the `group.id` will determine which Workers will cooperate as a team.

Connect Worker Groups will use unique administrative topics to hold connector configurations, offsets for both source and sinks, and status (if necessary) of the external data sources/sinks. Different worker groups can use different administrative topics if you need to keep their configurations separate.

	Make sure that all Workers with the same <code>group.id</code> are using the same names for the administrative topics or you will get unpredictable results.
---	--

# Creating Kafka Connect's Topics Manually

- The Topics used in Kafka Connect distributed mode will be **automatically created**
- They are created with **recommended values** for
  - replication factor
  - partition counts
  - cleanup policy

The following number of Partitions are created by default for each Topic:

Topic	Partitions
config.storage.topic	1
offset.storage.topic	25
status.storage.topic	5

Notice that the recommendations for `offset.storage.topic` are similar to the `__consumer_offsets` Topic since they do similar jobs.



In general, it's usually ok to let Connect create these topics for us. Connect will create these topics with the recommended replication factor, partition counts, and cleanup policy.

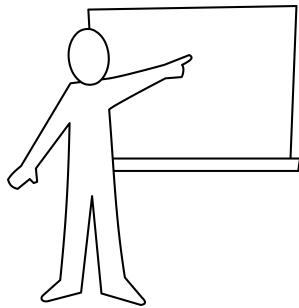
Settings for these internal topics can be customized using the following configuration settings:

```
config.storage.replication.factor offset.storage.replication.factor  
status.storage.replication.factor offset.storage.partitions  
status.storage.partitions config.storage.<topic-specific-setting>  
offset.storage.<topic-specific-setting> status.storage.<topic-specific-setting>
```

For additional information, see

<https://docs.confluent.io/current/connect/userguide.html#distributed-worker-configuration>

# Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors ... ↵
- Single Message Transforms (SMTs)
- Examples
-  Hands-on Lab: **Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# Configuring Connectors with Confluent Control Center

The left screenshot shows the 'Browse' page in the Confluent Control Center. It displays a list of connectors under the 'Connect' category, including 'ActiveMQSource Connector', 'JdbcSource Connector', and 'FileStreamSink Connector'. Each connector has a 'Source' button and a 'Connect' button. The right screenshot shows the 'Add Connector' page for a JDBC source. It has two tabs: '01 SETUP CONNECTION' and '02 TEST AND VERIFY'. Under '01 SETUP CONNECTION', there are fields for 'Connector class' (set to 'io.confluent.connect.jdbc.JdbcSourceConnector'), 'name' (set to 'My-JDBC-Source'), and 'Common' settings like 'Tasks max' (set to '3'). A sidebar on the right provides links to 'How should we connect to your data?' and other configuration options such as 'Common', 'Transforms', 'Error Handling', 'Database', 'Mode', 'Connector', and 'Additional Properties'.

Connectors can also be configured in Confluent Control Center. We can define source and sink connectors there.



Not all connectors may be fully configurable in the Control Center. In this case one has to use the Connect REST API to create and configure the connector. In the hands-on lab we give an example of this.

# Configuring Connectors with the REST API

- Add, modify delete connectors
- Distributed Mode:
  - Config **only** via REST API
  - Config stored in Kafka topic
  - REST call to **any** worker
- Standalone Mode:
  - Config also via REST API
  - Changes **not persisted!**
- Control Center uses REST API

- 
- Connectors can be added, modified, and deleted via a REST API on port 8083
  - In distributed mode, configuration can be done only via this REST API
    - Changes made this way will persist after a worker process restart
    - Connector configuration data is stored in a special Kafka Topic
    - The REST requests can be made to any worker
  - In standalone mode, configuration can also be done via a REST API
    - However, typically configuration is done via a properties file
      - Changes made via the REST API when running in standalone mode will not persist after worker restart
  - Confluent Control Center leverages this REST API to let users configure and manage connectors through the GUI

To make changes to connectors, the Worker will also run a REST API as part of its code. This allows HTTP calls to be sent to any of the Workers that will then configure the connectors. Changes made via the REST API take effect immediately and without a reboot.



The REST API included with a Connect Worker is different from the Confluent REST Proxy.

# Using the REST API

Some important REST endpoints

Method	Path	Description
GET	/connectors	Get a list of active connectors
POST	/connectors	Create a new Connector
GET	/connectors/(string: name)/config	Get configuration information for a Connector
PUT	/connectors/(string: name)/config	Create a new Connector, or update the configuration of an existing Connector

At times a requirement exists for the Connect REST API to return certain HTTP headers. The `response.http.headers.config` setting can be used to customize HTTP response headers.

Example:

```
response.http.headers.config="add Cache-Control: no-cache, no-store, must-revalidate",  
add X-XSS-Protection: 1; mode=block, add Strict-Transport-Security: max-age=31536000;  
includeSubDomains, add X-Content-Type-Options: nosniff
```

Output of Response Header:

```
< HTTP/1.1 200 OK  
< Date: Sat, 07 Mar 2020 17:33:39 GMT  
< Strict-Transport-Security: max-age=31536000;includeSubDomains  
< X-XSS-Protection: 1; mode=block  
< X-Content-Type-Options: nosniff  
< Cache-Control: no-cache, no-store, must-revalidate  
< Content-Type: application/json  
< Vary: Accept-Encoding, User-Agent  
< Content-Length: 136
```

More information on the REST API can be found at

<https://docs.confluent.io/current/connect/references/restapi.html>

# Configuring the Connector

Property	Description
<code>name</code>	Connector's unique name
<code>connector.class</code>	Connector's Java class
<code>tasks.max</code>	Maximum tasks to create. The Connector may create fewer if it cannot achieve this level of parallelism (Default: 1)
<code>key.converter</code>	(optional) Override the worker key converter
<code>value.converter</code>	(optional) Override the worker value converter
<code>topics</code> (Sink connectors only)	List of input topics to consume from

- The number of tasks that a Connector should start will be dependent on the number of cores and workers available.
- Additional converters can be installed via Confluent Hub (when available).
- The reason that the `topics` property only exists for sink connectors is that source connectors typically have logic built in that will create new topic names based on a specific naming convention.

With the release of Kafka 2.6, it is now possible to define custom topic configurations for the topics that are created by source connectors using the following properties:

Property	Description
<code>topic.creation.groups</code>	A list of group aliases that will be used to define per group topic configurations for matching topics. The group <code>default</code> always exists and matches all topics.
<code>topic.creation.\$alias.include</code>	Regular expressions that identify topics to include.
<code>topic.creation.\$alias.exclude</code>	Regular expressions that identify topics to exclude.
<code>topic.creation.\$alias.replication.factor</code>	$\geq 1$ for a specific valid value, or -1 to use the broker's default value
<code>topic.creation.\$alias.partitions</code>	$\geq 1$ for a specific valid value, or -1 to use the broker's default value
<code>topic.creation.\$alias.\${kafkaTopicSpecificConfigName}</code>	List of input topics to consume from

Different classes of configuration properties can be defined through the definition of groups.

## Example:

Portion of an example source connector configuration using topic creation rules

- By default, new topics created by Connect for this connector will have replication factor of 3 and 5 partitions
- Topics that match the inclusion list of the *inorder* group which will have 1 partition

```
...
topic.creation.groups=inorder
topic.creation.default.replication.factor=3
topic.creation.default.partitions=5

topic.creation.inorder.include=status, orders.*
topic.creation.inorder.partitions=1
```

For additional information, see KIP-158.

<https://cwiki.apache.org/confluence/display/KAFKA/KIP-158%3A+Kafka+Connect+should+allow+source+connectors+to+set+topic-specific+settings+for+new+topics>

tejaswin.renugunta@walgreens.com

# Overriding Connect Worker Properties

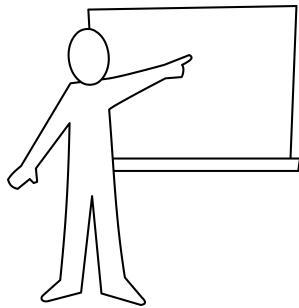
- By default, a worker's configurations apply to **all** connectors running on the worker
- Customize a connector's settings with `connector.client.config.override.policy`
  - Overrides worker configurations with `producer`, `consumer`, or `admin` prefixes
  - **None** (default): use worker settings
  - **Principal**: override security settings
  - **All**: override any prefixed settings
  - You can create your own custom `ConnectorClientConfigOverridePolicy` class
- Granular control over security and performance for individual connectors

---

Apache Kafka 2.3 and CP 5.3 introduced the ability for individual connectors to override the worker-level configurations with the `connector.client.config.override.policy` property. For more information about `connector.client.config.override.policy`, see <https://docs.confluent.io/current/connect/security.html#separate-principals>

tejaswin.renugunta@wileycs.com

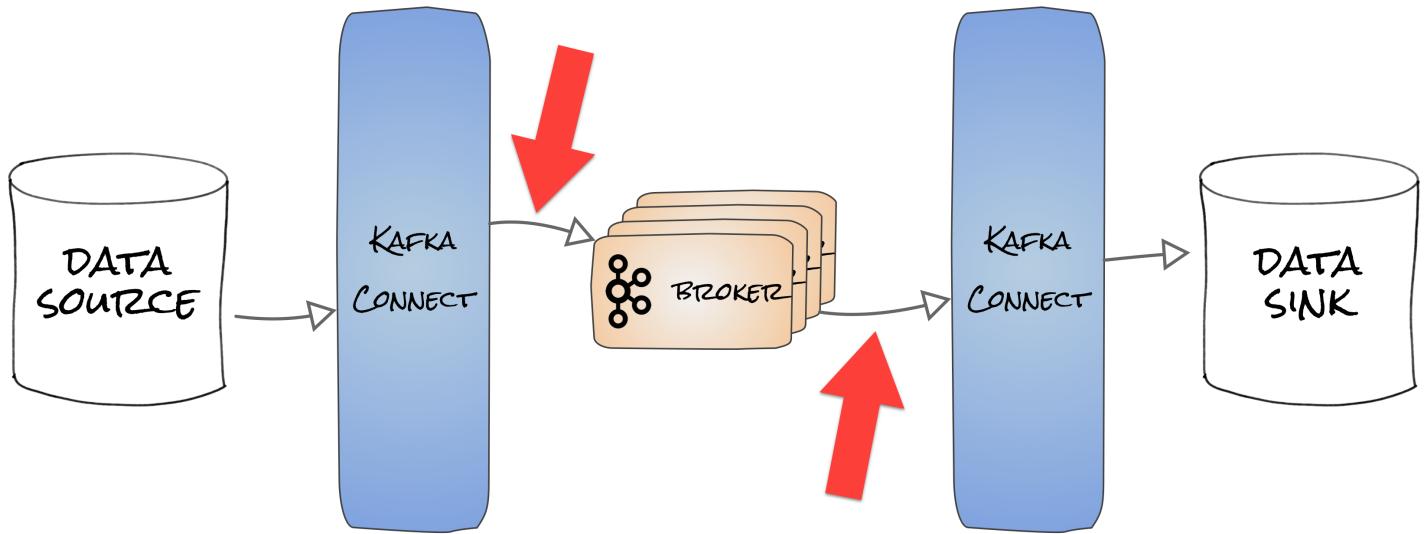
# Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs) ... ↵
- Examples
-  Hands-on Lab: **Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# Transforming Data with Connect



Now, recall the architecture. Imagine we wanted to change messages in there.

There are at least two options discussed on the following slides...

# Single Message Transform (SMT)

Modify events before storing in Kafka:

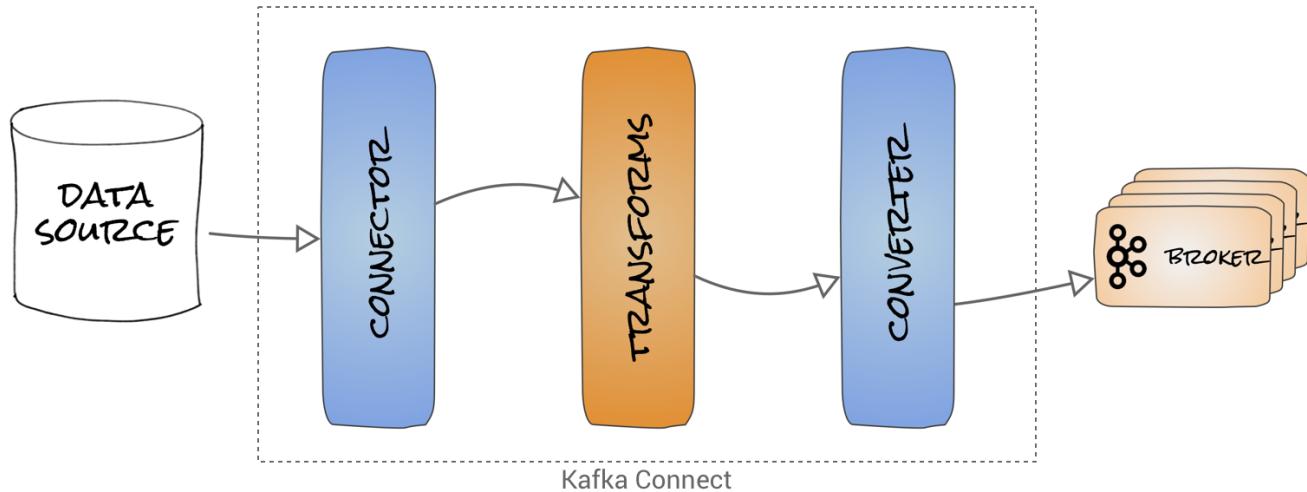
- Mask sensitive information
- Add identifiers
- Tag events
- Lineage/provenance
- Remove unnecessary columns

Modify events going out of Kafka:

- Route high priority events to faster data stores
- Direct events to different Elasticsearch indices
- Cast data types to match destination
- Remove unnecessary columns

tejaswin.renugunta@walgreens.com

# Where SMTs Live



Single message transforms are another stage in the Kafka Connect pipeline. Here we can see where they fit in for a source connector.

Source connectors read data from another system and store the data in Kafka. The connector (or more accurately, the connector's tasks) read data from the data source. At this stage the data will be in whatever format the source system exposes, but the connector will translate that into Kafka Connect's generic data API. Once it is in this generic format, the Kafka Connect single message transformations can be applied. Because they use the generic data format, any transformation can be used with any connector, independent of the original format of the data. The transformation is applied and the resulting data is still in a generic data API format. Finally, the data is passed to a converter, which serializes it into a format like Avro or JSON before it is passed to Kafka for storage. The pipeline for sink connectors looks similar, just reversed.

As you can see, by breaking the processing down into separate steps and leveraging Connect's data API, we can make transformations very general and applicable regardless of which connector you're using, much like we make serialization formats reusable across all connectors.

# Single Message Transform - Details (1)

Transform	Description
InsertField	insert a field using attributes from message metadata or from a configured static value
ReplaceField	rename fields, or apply a blacklist or whitelist to filter
MaskField	replace field with valid <b>null</b> value for the type (0, empty string, etc)
ValueToKey	replace the key with a new key formed from a subset of fields in the value payload
HoistField	wrap the entire event as a single field inside a <b>Struct</b> or a <b>Map</b>
ExtractField	extract a specific field from <b>Struct</b> and <b>Map</b> and include only this field in results
SetSchemaMetadata	modify the schema name or version
TimestampRouter	modify the topic of a record based on the original topic name and timestamp
RegexRouter	update a record topic using the configured regular expression and replacement string

tejaswin.renugunta@walgreens.com

# Single Message Transform - Details (2)

Transform	Description
Flatten	flatten a nested data structure, generating names for each field by concatenating the field names at each level with a configurable delimiter character.
Cast	cast fields or the key/value to a specific type (e.g. to force an integer field to a smaller width)
TimestampConverter	convert timestamps between different formats
Filter	conditionally to filter out records matching (or not matching) a particular predicate
HasHeaderKey	a predicate which is true for records with at least one header with the configured name
RecordIsTombstone	a predicate which is true for records which are tombstones (i.e. have null value)
TopicNameMatches	a predicate which is true for records with a topic name that matches the configured regular expression

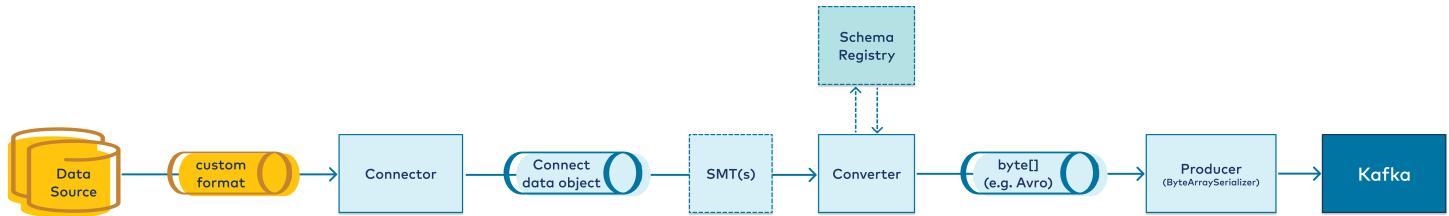
Many customers need a way to perform simple filtering and transformations for which building a streams application is too much work. For example, a company might want to take order information and send that to a team for analysis but needed to redact sensitive information such as social security numbers or credit card numbers.

The above properties can be configured at the **connector level** and be applied to **key** and/or **value**.

More info: [http://kafka.apache.org/documentation.html#connect\\_transforms](http://kafka.apache.org/documentation.html#connect_transforms)

# Converting Data

- Converters provide the data format written to or read from Kafka (like Serializers)
- Converters are decoupled from Connectors so they can be reused
  - Allows any connector to work with any serialization format
- Example of format conversion in a source converter (sink converter is the reverse)



Serialization formats may seem like a minor detail, but **not** separating the details of data serialization in Kafka from the details of source or sink systems would result in a lot of inefficiency:

- First, a lot of code for doing simple data conversions are duplicated across a large number of ad hoc connector implementations.
- Second, each connector ultimately contains its own set of serialization options as it is used in more environments—JSON, Avro, Thrift, ProtoBuf, and more.

Much like the serializers in Kafka's producer and consumer, the **Converters** abstract away the details of serialization. Converters are different because they guarantee data is transformed to a common data API defined by Kafka Connect. This API supports both schema and schema-less data, common primitive data types, complex types like structs, and logical type extensions.

By sharing this API, connectors write one set of translation code and Converters handle format-specific details. For example, the JDBC connector can easily be used to produce either JSON or Avro to Kafka, without any format-specific code in the connector.

# Converter Data Formats

- Converters apply to both the key and value of the message
  - Key and value converters can be set independently
    - `key.converter`
    - `value.converter`
- Pre-defined data formats for Converter
  - Avro: `AvroConverter`
  - Byte Array: `ByteArrayConverter`
  - JSON: `JsonConverter`
  - JSON Schema: `JsonSchemaConverter`
  - Protobuf: `ProtobufConverter`
  - String: `StringConverter`

The `ByteArrayConverter` (detailed in KIP-128, introduced in Kafka 0.11) provides an option for dealing with raw data, in contrast to structured data. It saves cost (CPU, memory, garbage collection overhead) of deserializing/converting data to Connect format.

For example, this is particularly useful with **Replicator**. This connector doesn't need to convert to/from byte array format and objects - it just copies the raw data as bytes for better performance.

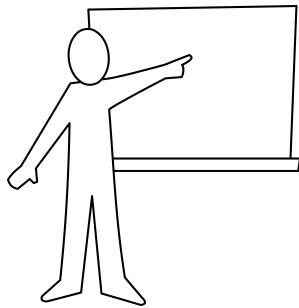
# Converter Best Practice

- Use a converter that supports Schema Registry!
  - Avro, JSON Schema, and Protobuf converters
- This gives connectors all the Schema Registry benefits that regular producers and consumers enjoy:
  - Planned schema evolution
  - Schemas durably persisted in Kafka
  - Each messages sent with schema id instead of whole schema

*connect-distributed.properties*

```
...
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://schemaregistry1:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://schemaregistry1:8081
...
```

# Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples ... 
-  Hands-on Lab: **Running Kafka Connect**

tejaswin.renugunta@walgreens.com

# Example: JDBC Source Connector

- Java Database Connectivity (JDBC) API is common amongst databases
  - JDBC Source Connector is a great way to get database tables into Kafka topics
  - JDBC Source periodically polls a relational database for new or recently modified rows
    - Creates a record for each row, and Produces that record as a Kafka message
  - Each table gets its own Kafka topic
  - New and deleted tables are handled automatically
- 

The JDBC source connector allows you to import data from any relational database with a JDBC driver into Kafka topics. By using JDBC, this connector can support a wide variety of databases without requiring custom code for each one.

Data is loaded by periodically executing a SQL query and creating an output record for each row in the result set. By default, all tables in a database are copied, each to its own output topic. The database is monitored for new or deleted tables and adapts automatically. When copying data from a table, the connector can load only new or modified rows by specifying which columns should be used to detect new or modified data.

# Query Mode (1)

Incremental query mode	Description
Incrementing column	Check a single column where newer rows have a larger, auto-incremented ID. Does not capture updates to existing rows.
Timestamp column	Checks a single 'last modified' column to capture new rows and updates to existing rows. If task crashes before all rows with the same timestamp have been processed, some updates may be lost.
Timestamp and incrementing column	Detects new rows and updates to existing rows with fault tolerance. Uses timestamp column, but reprocesses current timestamp upon task failure. Incrementing column then prevents duplicate processing.

The Connector can detect new and updated rows in several ways as described on the slide.

For the reasons stated on the slides, many environments will use both the timestamp and the incrementing column to capture all updates.

Because timestamps are not necessarily unique, the timestamp column mode cannot guarantee all updated data will be delivered. If two rows share the same timestamp and are returned by an incremental query, but only one has been processed before the Connect task fails, the second update will be missed when the system recovers.

## Query Mode (2)

Incremental query mode	Description
Custom query	Used in conjunction with the options above for custom filtering.
Bulk	Load all rows in the table. Does not detect new or updated rows.

The custom query option can only be used in conjunction with one of the other incremental modes as long as the necessary WHERE clause can be appended to the query. In some cases, the custom query may handle all filtering itself.



Use **bulk** mode for one-time load, not incremental, unfiltered

# Configuration

Property	Description
<code>connection.url</code>	The JDBC connection URL for the database
<code>topic.prefix</code>	The prefix to prepend to table names to generate the Kafka Topic name
<code>mode</code>	The mode for detecting table changes. Options are <code>bulk</code> , <code>incrementing</code> , <code>timestamp</code> , <code>timestamp+incrementing</code>
<code>query</code>	The custom query to run, if specified
<code>poll.interval.ms</code>	The frequency in milliseconds to poll for new data in each table (Default: 5000)
<code>table.blacklist</code>	A list of tables to ignore and not import. If specified, <code>tables.whitelist</code> cannot be specified
<code>table.whitelist</code>	A list of tables to import. If specified, <code>tables.blacklist</code> cannot be specified



See [JDBC Connector docs](#) for a complete list

Setting both `table.whitelist` and `table.blacklist` does not fail any upfront configuration validation checks but will fail when starting the connector at runtime.

# JDBC Source Connector with SMTs (1)

```
1 {
2   "name": "Driver-Connector",
3   "config": {
4     "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
5     "connection.url": "jdbc:postgresql://postgres:5432/postgres",
6     "connection.user": "postgres",
7     "table.whitelist": "driver",
8     "topic.prefix": "",
9     "mode": "timestamp+incrementing",
10    "incrementing.column.name": "id",
11    "timestamp.column.name": "timestamp",
12    "table.types": "TABLE",
13    "numeric.mapping": "best_fit",
```

The goal of this connector is to take the `driver` table of a Postgres database and produce its records to Kafka. We would like each Kafka record to have a string key for the driver ID ("driver-1", "driver-2", etc.). We would also like the value of each Kafka record to be an avro record with id, driverkey, firstname, lastname, make, model, and timestamp.

Unfortunately, the configurations shown will not result in the schema we want. First, the topic name would be "driver" rather than "driver-profiles-avro". Second, the record keys would be `NULL` and the values would include a field that looks like `{"driverkey": "driver-3"}`. We can modify these minor details using SMTs, which is shown in the next slide.

# JDBC Source Connector with SMTs (2)

```
14  "transforms": "suffix,createKey,extractKey",
15  "transforms.suffix.type": "org.apache.kafka.connect.transforms.RegexRouter",
16  "transforms.suffix.regex": "(.*)",
17  "transforms.suffix.replacement": "$1-profiles-avro",
18  "transforms.createKey.type": "org.apache.kafka.connect.transforms.ValueToKey",
19  "transforms.createKey.fields": "driverkey",
20  "transforms.extractKey.type":
21      "org.apache.kafka.connect.transforms.ExtractField$Key",
22  "transforms.extractKey.field": "driverkey"
23 }
24 }
```

The connector takes the `driver` table of a Postgres database and produces its records to Kafka. We would like each Kafka record to have a string key for the driver ID ("driver-1", "driver-2", etc.). The value of each Kafka record will be an avro record with id, driverkey, firstname, lastname, make, model, and timestamp.

- In line 14, we define three transformations: `suffix`, `createKey`, and `extractKey`. These names can be anything, but it is recommended that they succinctly describe the transformations.
- In line 15, we define the class that will be used for the `suffix` transformation. In this case, we use the `RegexRouter` class, which is a class that sets the Kafka topic name. Normally, the topic name would be "prefix" + "table". Earlier, we set `topic.prefix` to the empty string. So the topic name should just be the name of the table, which is "driver". This transformation replaces "driver" with "driver-profiles-avro".
- In line 18, we define the class used for the `createKey` transformation. In this case, we use the `ValueToKey` class, which is a class that replaces the default Kafka record key with a new key from a field in the table. In this case, we use the `driverkey` field as the key. Without this transformation, the keys would be `null`. With this transformation, an example key would be `{"driverkey": "driver-3"}`.
- In line 20, we further refine the key with the `ExtractField$Key` class. We extract the string associated with `driverkey`. Before this transformation, an example Kafka record key would be `{"driverkey": "driver-3"}`. After this transformation, the Kafka record key is simply the string "driver-3".
- In the labs, the Connect workers are configured to use the string converter for keys and the avro converter for values. So the key will be properly serialized as a string and the value will be serialized as an avro record consisting of all the fields from the table. The Connect workers are also configured to use the Schema Registry, so this schema will be available to view at `schema-registry:8081/subjects/driver-profiles-avro-value/versions/1`. We will discuss converters in more detail in the next few slides.

For more information on SMTs, see

<https://docs.confluent.io/current/connect/transforms/index.html>

tejaswin.renugunta@walgreens.com

# Example: HDFS Sink Connector

- Continuously polls from Kafka and writes to HDFS (Hadoop Distributed File System)
- Integrates with Hive
  - Auto table creation
  - Schema evolution with Avro
- Works with secure HDFS and the Hive Metastore, using Kerberos
- Provides exactly once delivery
- Data format is extensible
  - Avro, Parquet, custom formats
- Pluggable Partitioner, supporting:
  - Kafka Partitioner (default)
  - Field Partitioner
  - Time Partitioner
  - Custom Partitioners

---

The HDFS connector allows you to export data from Kafka topics to HDFS files in a variety of formats and integrates with Hive to make data immediately available for querying with HiveQL.

The connector periodically polls data from Kafka and writes them to HDFS. The data from each Kafka topic is partitioned by the provided partitioner and divided into chunks. Each chunk of data is represented as an HDFS file with topic, kafka partition, start and end offsets of this data chunk in the filename. If no partitioner is specified in the configuration, the default partitioner which preserves the Kafka partitioning is used. The size of each data chunk is determined by the number of records written to HDFS, the time written to HDFS and schema compatibility.

The HDFS connector integrates with Hive and when it is enabled, the connector automatically creates an external Hive partitioned table for each Kafka topic and updates the table according to the available data in HDFS.

# Example: FileStream Connector

- Great tool for learning about connectors
  - FileStream Connector acts on a local file
    - Local file Source Connector: tails local file and sends each line as a Kafka message
    - Local file Sink Connector: Appends Kafka messages to a local file
- 

The FileSource Connector reads data from a file and sends it to Kafka. This connector will read only one file and send the data within that file to Kafka. It will then watch the file for appended updates only. Any modification of file lines already sent to Kafka will not be reprocessed.

The FileSink Connector reads data from Kafka and outputs it to a local file. Multiple topics may be specified as with any other sink connector. As messages are added to the topics specified in the configuration, they are produced to a local file as specified in the configuration.

The FileStream Connector examples on [docs.confluent.io](https://docs.confluent.io) are intended to show how a simple connector runs for those first getting started with Kafka Connect as either a user or developer. It is not recommended for production use. Instead, we encourage users to use them to learn in a local environment. The examples include both a file source and a file sink to demonstrate an end-to-end data flow implemented through Kafka Connect. The FileStream Connector examples are also detailed in the developer guide as a demonstration of how a custom connector can be implemented.

# Hands-On Lab

- In this exercise, you will run Connect in distributed mode, and use the JDBC source Connector and File sink Connector. You will configure monitors using the REST API as well as Confluent Control Center. You will use Confluent Control Center to monitor the connectors as well.
- Please refer to **Lab 07 Data Pipelines with Kafka Connect** in the Exercise Book:
  - a. **Running Kafka Connect**



tejaswin.renugunta@walgreens.com

# Module Review

## Questions:

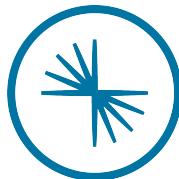


- Why would you use Kafka Connect?
- What kind of supported connectors are you aware of?
- What if no connector exists for your source or sink?

- Kafka Connect provides a scalable, reliable way to transfer data from external systems into Kafka, and vice versa
- Many off-the-shelf Connectors are provided by Confluent, and many others are under development by third parties
- The workflow for creating a new connector is well documented [here](#). Write a Java application that implements pre-defined interfaces and register the resulting JAR with Kafka Connect.

tejaswin.renuguntangreens.com

# 08: Kafka in Production



CONFLUENT

tejaswin.renugunta@walgreens.com

# Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Providing Durability
4. Managing a Kafka Cluster
5. Optimizing Kafka's Performance
6. Kafka Security
7. Data Pipelines with Kafka Connect
8. Kafka in Production ... ←
9. Conclusion

tejaswin.renugunta@walgreens.com

# Learning Objectives

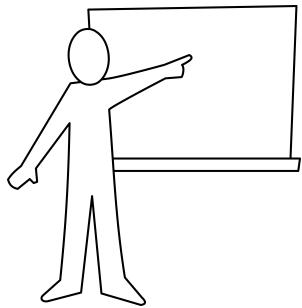


After this module you will be able to:

- Deploy highly available services:
  - Kafka
  - ZooKeeper
  - Kafka Connect
  - Confluent Schema Registry
  - Confluent REST Proxy
  - Kafka Streams and ksqlDB
  - Confluent Control Center
- Do capacity planning
- List key considerations when deploying Kafka in multiple data centers

tejaswin.renugunta@walgreens.com

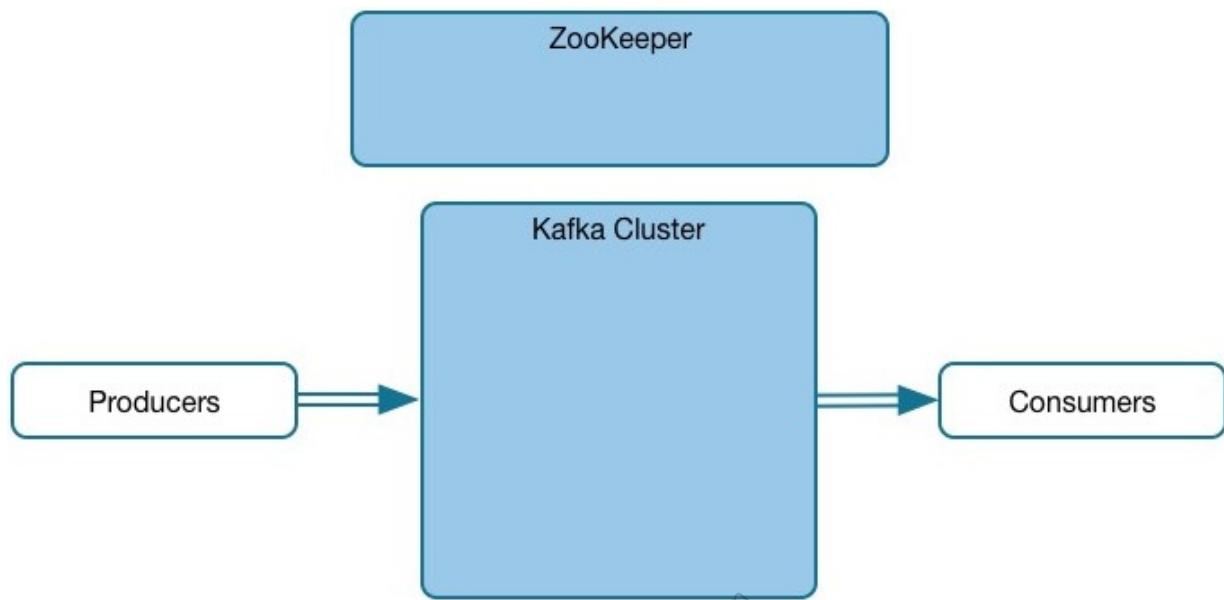
# Module Map



- Typical Kafka Design Model ... ←
- Brokers
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers

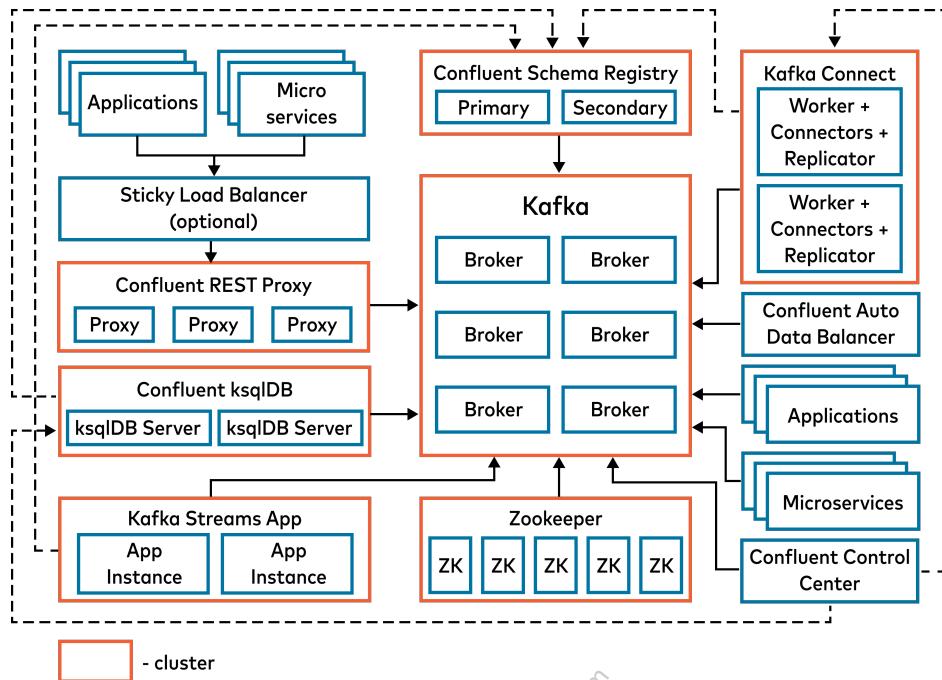
tejaswin.renugunta@walgreens.com

# Kafka Basic Pub/Sub



This diagram shows the minimum architecture that is required for Kafka as a pub/sub message queue.

# Kafka Reference Architecture



While Kafka began as a highly durable, high-throughput message queue, it has grown into an ecosystem that can act as the central nervous system of an event-driven business. This production-level architecture was built to scale. Each component is given its own servers (as noted by blue boxes), and if any layer becomes overloaded, it can be scaled independently simply by adding nodes to that specific layer. For example, when adding applications that use the Confluent REST Proxy, you may find that the REST Proxy no longer provides the required throughput while the underlying Kafka Brokers still have spare capacity. In that case, you only need to add REST Proxy nodes in order to scale your entire platform.

The image might be overwhelming at first. Assure students that they will break down this architecture piece-by-piece and study deployment considerations for each component. Consider asking students:

- What do you notice?
- What do you wonder?

Give students a few minutes to write down their thoughts before asking them to share with peers and share in a brief, whole-class discussion. Some features to briefly note if not mentioned by students:

- The orange boxes surround Kafka components that are run in clusters.
- Confluent REST Proxy requires a sticky load balancer.

- Microservices/applications can access Kafka either directly (using Kafka client APIs), or via Confluent REST Proxy.
- Confluent Schema Registry has a Primary/Secondary architecture.
- Confluent Auto Data Balancer is executed on Brokers.
- Confluent's multi data center replication tool Replicator is itself a specialized Connector and thus is deployed in the Kafka Connect cluster in the *destination* cluster.

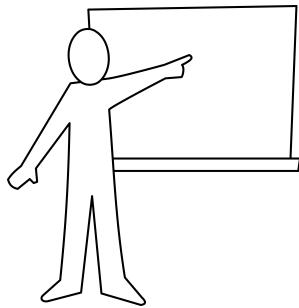
Each of these points will be discussed in more detail later, so don't worry if there is not enough time to discuss these details.



Not shown in the diagram is the fact that Producer/Consumer applications also interact with Confluent Schema Registry, not just Kafka Connect and ksqlDB. This will also be discussed in more detail later.

tejaswin.renugunta@walgreens.com

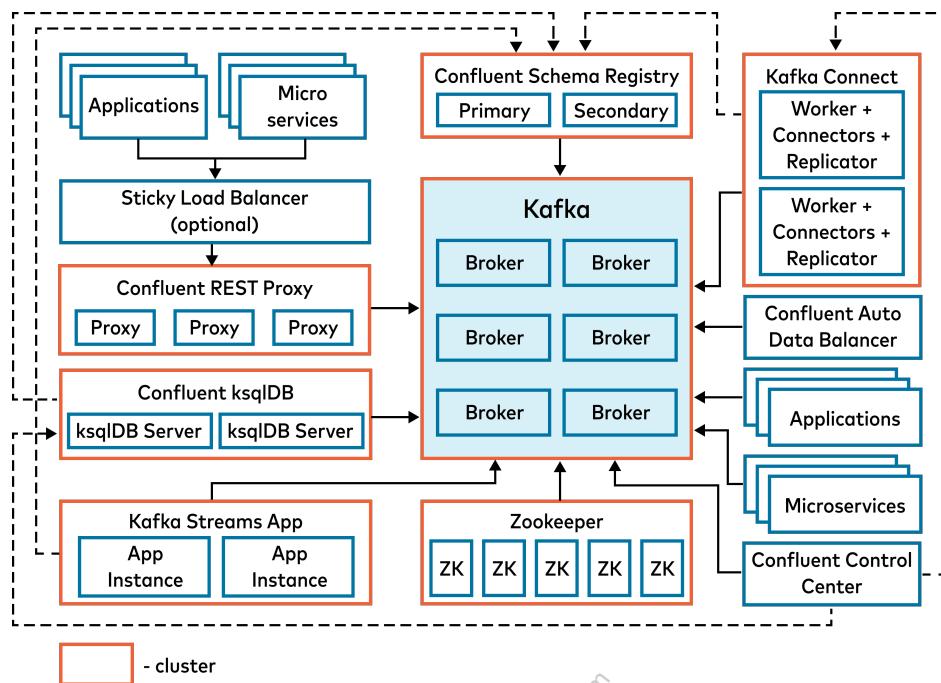
# Module Map



- Typical Kafka Design Model
- Brokers ... ←
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers

tejaswin.renugunta@walgreens.com

# Kafka Reference Architecture: Brokers



tejaswin.renugunta@walgreens.com

# Broker Design

- Run on dedicated servers
- $N$  Brokers  $\rightarrow$  replication factor up to  $N$
- Can use virtual IP + load balancer as `bootstrap.servers`
  - Pro: don't have to change client config code when cluster changes
  - Con: More infrastructure to worry about
- Discussion Questions:
  - What replication factor is acceptable for mission critical data?
  - How many Brokers do you need to accommodate this?
  - With that many Brokers, how many can fail without permanent data loss? What would happen to write access and read access?

---

Best practice is to separate Brokers and ZooKeeper nodes, but it is not uncommon to combine them on a single server in small environments. If this is the case, it is recommended to use separate disks for ZooKeeper data.

When configuring clients, a property `bootstrap.servers` is required. This Broker list is only used for the initial metadata pull when the client initializes (hence the "bootstrap" part of the name). Once the first metadata response is received, the Producer will send produce requests to the Broker hosting the corresponding Topic/Partition directly, using the IP/port the Broker registered in ZooKeeper. For metadata update requests, the client can contact any Broker. The `bootstrap.servers` property should list multiple Brokers so that a single offline Broker does not prevent the client initialization. An alternative is to use a VIP in a load balancer. If Brokers change in a cluster, one can just update the hosts associated with the VIP.

The discussion questions are review of earlier durability concepts. Mission critical Topics should have replication factor 3 or greater. Replication factors higher than 5 or so begin to have diminishing returns because the probability of simultaneous data loss decreases exponentially with the number of replicas, while the cost of provisioning more storage increases linearly. Assuming a replication factor of 3 for mission critical Topics, there would need to be at least 3 Brokers—2 of which could fail without resulting in permanent data loss. However, if only 2 Brokers are available, then new Topics with replication factor 3 cannot be created. Write access may also be blocked depending on the `min.insync.replicas` Topic configuration. For these reasons, it is usually a good idea to have *more* Brokers than the highest expected replication factor. Read access will continue, but perhaps there will be downtime while new leader replicas are elected.



If using SASL GSSAPI, virtual IPs present a challenge since authentication has to happen against the actual Broker hostname/IP address. It may be possible to configure the load balancers and Kerberos to handle this scenario, but that is beyond the scope of this course.



Apache Kafka 2.1.0 and KIP-302 introduced the `use_all_dns_ips` option for the `client.dns.lookup` client property. KIP-602 changed the default value for `client.dns.lookup` to be `use_all_dns_ips` so that it will attempt to connect to the broker using all of the possible IP addresses of a hostname. The default is intended to reduce connection failure rates and is more important in cloud and containerized environments where a single hostname may resolve to multiple IP addresses.

tejaswin.renugunta@walgreens.com

# Capacity Planning: Brokers

- Disk space and I/O
- Network bandwidth
- RAM (for page cache)
- CPU

---

The most heavily used resource in a Broker is disk space. Kafka commit logs store a large amount of data on disk.



Prior to Kafka 1.0.0, a single disk failure caused a hard failure of the Broker. If the Broker got an i/o exception when doing a write, it would kill itself. Kafka 1.1 added the `kafka-reassign-partitions` tool to help balance disk utilization when mounting separate disks to log directories:  
<https://cwiki.apache.org/confluence/display/KAFKA/KIP-113%3A+Support+replicas+movement+between+log+directories>

The second most heavily used resource is network bandwidth. Every message is typically used multiple times. At LinkedIn, the ratio is 1:5 - each message is read five times, including internal replication.

Brokers do not need a lot of RAM for the JVM heap space since the Broker does not cache messages there. Kafka relies on the page cache for its commit logs so the more RAM you have on the Broker, the more buffer space will be available for those messages.

Brokers are not CPU-intensive by default, but Kafka is highly multithreaded, so Brokers will benefit more from many cores than from faster cores.

# Broker Disk

- 12 x 1TB filesystems mounted to data directories for Topic data + separate disks for OS
  - A single Partition can only live on **one** volume
  - Partitions are balanced across **log.dirs** in round-robin
  - RAID-10 optional
- Use XFS or EXT4 filesystems
  - Mount with **noatime**



---

- Discussion Questions:

- What are the pros and cons of using RAID-10 rather than raw disks?
  - What are the pros and cons of using distributed storage for Kafka logs?

---

A typical production grade Broker has 6-12 1TB disks for Topic data (and perhaps RAID-1 on the OS volume for fault-tolerance). The exact amount of storage you will need depends on the number of Topics, Partitions, the rate at which applications will be writing to each Topic, and the retention policies you configure.

Kafka can use either EXT4 or XFS for the mounted file systems. Some environments have reported better performance with XFS.

Access time, or **atime**, is the way that Linux maintains file system metadata that records when each file was last accessed. As a result, every read operation on a filesystem is not just a read operation; it is also a write operation since the **atime** needs to be updated. **noatime** disables **atime** collection and improves performance.

There are two strategies for local storage:

1. Put a filesystem on each physical disk and mount one filesystem to each log directory listed in **log.dirs** (Broker property), or
2. Use RAID-10 (stripe, then mirror)

Here are some ideas to bring up during discussion if not mentioned by students:

- Local storage:
  - Using multiple physical disks provides means a single disk failure will not cause the Broker to fail. Just replace that disk and Kafka's automatic replication will recover the data (although this will impact network bandwidth).
  - If you list multiple directories in `log.dirs`, the Broker will assign Partitions using round-robin. If some Partitions are bigger than others, you'll end up with uneven disk usage. This would need to be monitored and manually balanced with `kafka-reassign-partitions`. As of this writing, The Confluent Auto Data Balancer does not yet have the ability to automatically balance storage load between log directories on individual Brokers.
  - RAID-10 combines striping and mirroring. Striping gives much better write performance while mirroring protects a single disk failure from taking out the whole RAID array. RAID systems also typically have additional benefits such as online resizing and hot swappable disks to minimize system downtime. If the RAID array is a single filesystem, then you don't have to worry about unevenly distributed data. The downside of RAID-10 is that it requires at least twice as much storage, depending on how disks are mirrored.
- Distributed storage:
  - Distributed storage like SAN and NAS can severely and adversely impact Kafka's performance and availability.
  - Cloud storage solutions like Elastic Block Storage from AWS can offer good enough performance to use as a log directory, and the durability guarantees that come with cloud storage may be enough for some customers to consider lowering replication factor for Topics within the Kafka cluster. This will be discussed in more detail later in the module.

# Network Bandwidth

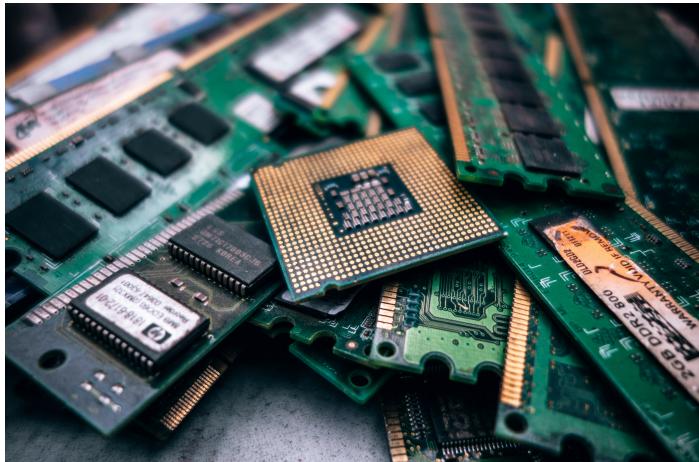
- Gigabit Ethernet sufficient for smaller applications
  - 10Gb Ethernet needed for large installations
  - Enable compression on Producers
  - Optional: Isolate Broker-ZooKeeper traffic to separate network
- 

When provisioning for network capacity, you will want to take into account replication traffic between Brokers and leave some overhead for rebalancing operations and bursty clients. Network is one of the resources that is most difficult to provision since adding nodes will eventually run against switch limitations. Therefore, consider enabling compression to get better throughput from existing network resources. Note that a Kafka Producer will compress messages in batches, so configuring the Producer to send larger batches will result in a better compression ratio and improved network utilization.

With the removal of ZooKeeper requirements from most of the components, isolating Broker-ZooKeeper traffic to a separate network is becoming more common. The benefits are twofold:

- Less contention on the network between Broker-Client and Broker-ZooKeeper traffic
- Isolation of ZooKeeper from clients (and users) for security since ACLs and passwords are kept in ZooKeeper

# Broker Memory



- JVM heap ~ 6 GB
- OS ~ 1 GB
- Page cache:
  - Lots and lots!
  - What might your Consumer lag be?

---

The Broker software itself does not have heavy memory requirements. Any RAM beyond what is needed for the OS and the JVM Heap will be available for page cache. The page cache is what gives Kafka such amazing performance; it allows for *zero copy transfer*, where data is sent directly to the network buffer without context switching between kernel space and user space.

The amount of memory used for the page cache depends on the rate that this data is written and how far behind you expect Consumers to get. If you write 20GB per hour per Broker and you allow Consumers to fall 3 hours behind in normal scenario, you will want to reserve 60GB to the OS page cache. In cases where Consumers are forced to read from disk, performance will drop significantly.

At *minimum*, a production Broker should have 32 GB of RAM. Typically they will use 64 GB or more.

# Tuning the Java Heap (1)

- Java Heap memory is allocated for storing Java objects
- By default `kafka-server-start` configures heap size to 1GB
  - `Xmx`: maximum Java heap size
  - `Xms`: start Java heap size

```
$ export KAFKA_HEAP_OPTS="-Xms6g -Xmx6g"
```

- Recommended JVM performance tuning options:

```
-Xms6g -Xmx6g -XX:MetaspaceSize=96m -XX:+UseG1GC -XX:MaxGCPauseMillis=20  
-XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M  
-XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
```

---

The JVM heap and garbage collection tuning recommendations given on this page are taken directly from <https://docs.confluent.io/current/kafka/deployment.html#jvm>. These were tested in a large deployment on JDK 8. It is recommended to use the G1 garbage collector, which is shown as `-XX:+UseG1GC`. The main options to concentrate on are `-Xms` and `-Xmx`. The 6 GB setting shown here is explained on the next slide. The `kafka-server-start` script uses the recommended options other than its default to 1 GB, so `KAFKA_HEAP_OPTS` will have to be set in production.

The `kafka-server-start` invokes a script called `kafka-run-class.sh` which takes options as environment variables. Here are several kinds of options that are declared with environment variables and translated in `kafka-run-class`:

- Heap options like `-Xms` and `-Xmx: $KAFKA_HEAP_OPTS`
  - Other performance tuning options like those on the slide:  
`$KAFKA_JVM_PERFORMANCE_OPTS`
- log4j application logging options: `KAFKA_LOG4J_OPTS`
- JMX options: `$KAFKA_JMX_OPTS`
- Metrics port: `$JMX_PORT`
- Generic JVM parameters like pointing to a JAAS file: `$KAFKA_OPTS`

# Tuning the Java Heap (2)

- Suggested formula for determining the Broker's heap size:

```
(message.max.bytes * num Partitions per Broker) + log.cleaner.dedupe.buffer.size +  
500MB
```

- Property defaults
  - `message.max.bytes` default is 1MB
  - `log.cleaner.dedupe.buffer.size` default is 128MB

- Consider tuning the Java heap size

Java Heap Size	Deployment Type
1 GB (default)	Testing and small production deployments
6 GB	Typical production deployments
12 GB+	Deployments with very large messages or many Partitions per Broker

Why is message size important when sizing the heap? As data is transferred between Brokers, the replication traffic will use the heap.

# Tuning Virtual Memory Settings

- Minimize memory swapping:
  - `vm.swappiness=1` (Default: 60)
- Decrease the frequency of blocking flushes (synchronous)
  - `vm.dirty_ratio=80` (Default: 20)
- Increase the frequency of non-blocking background flushes (asynchronous)
  - `vm.dirty_background_ratio=5` (Default: 10)



Set these parameters in `/etc/sysctl.conf` and load with `sysctl -p`

The native Linux mechanism of swapping processes from memory to disk can cause serious performance limitations in Kafka. Setting `vm.swappiness=1` prevents the system from swapping processes too frequently but still allows for emergency swapping instead of killing processes.

Configure when unflushed (*i.e.*, "dirty") memory is flushed to disk with `vm.dirty_background_ratio` and `vm.dirty_ratio`.

The `vm.dirty_ratio` is the highest percentage of memory that can remain unflushed before Linux blocks I/O. If the ratio is set low, these blocking flushes happen more frequently, which degrades Kafka's performance and prevents Consumers from benefiting from zero copy transfer. High ratios cause less frequent flushes, so we set the value to 80. Be aware that if a Broker has a sudden failure, all unflushed data is lost on that Broker. However, since production data is typically being replicated to other Brokers, this should not usually be a concern.

The `vm.dirty_background_ratio` is the percentage of system memory that can be dirty before the system can start writing data to disks in the background without blocking I/O. By decreasing this setting, we increase the frequency of non-blocking background flushes. On the one hand, we want to hold a large page cache to take advantage of zero copy transfer, but on the other hand, we also want to avoid building up too much dirty memory and forcing a blocking flush. These settings were tested extensively at LinkedIn, and it was found to be better to encourage more frequent background flushes and to discourage blocking flushes. Students are encouraged to run their own tests to monitor the effect of tuning these properties.



The recommendation used to be to set `vm.swappiness=0`. However, since RHEL 6.4 setting `vm.swappiness=0` more aggressively avoids swapping which increases the risk of OOM (out of memory) killing under strong memory and I/O pressure. Using `vm.swappiness=1` avoids this situation.

tejaswin.renugunta@walgreens.com

# Open File Descriptors and Client Connections

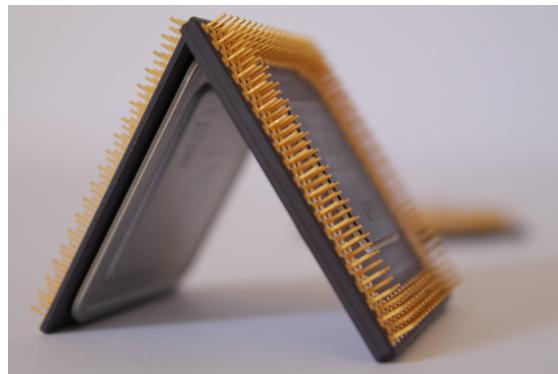
- Brokers can have a **lot** of open files
    - `$ ulimit -n 100000`
  - Brokers can have a **lot** of client connections:
    - `max.connections.per.ip` (Default: 2 billion)
- 

Linux limits the number of open files it can support to preserve system resources. The default setting is ~1000 or ~4000 open files, depending on the kernel. This is insufficient for a Broker due to the requirements of the Partitions. Each Partition requires a minimum of four file descriptors (socket, .log, .index., .timeindex), but typically maintains multiple sets of segment files depending on the roll settings. Since Brokers support a large number of Partitions, the number of open file handles could easily exceed the defaults, even in a relatively small deployment. The systemd unit files included in CP 5.3 automatically set the file descriptor limit to 100,000. To set this manually, use the `ulimit -n <large number>` command in Linux.

The `max.connections.per.ip` Broker property was added because of a real issue. A customer had a buggy, runaway application. Under some error conditions, the application created new connections to the Broker without closing old ones. Eventually, the Broker ran out of open file handles, causing the Broker to crash. As new leaders were elected for the Partition, the application repeated the process until it crashed the whole cluster. The `max.connections.per.ip` setting prevents this situation by limiting the number of connections a single IP address can make to a Broker. Tuning this setting is almost never necessary, but it is interesting to note its historical importance.

# Broker CPU

- Dual 12-core sockets
- Relevant Broker properties:
  - `num.io.threads` (Default: 8)
  - `num.network.threads` (Default: 3)
  - `num.recovery.threads.per.data.dir` (Default: 1)
  - `background.threads` (Default: 10)
  - `num.replica.fetchers` (Default: 1)
  - `log.cleaner.threads` (Default: 1)
- Discussion:
  - Given 12 data disks and dual 12-core CPU sockets, how would you modify the default Broker threading properties?



Most of the Confluent Platform components are not particularly CPU bound. If you notice high CPU, it is usually a result of misconfiguration, insufficient memory, or a bug. This is also true for Brokers, although Brokers are highly multithreaded and will benefit from tuning Broker thread properties, e.g., `background.threads`, `num.io.threads`, `num.network.threads`, `num.replica.fetchers`, `log.cleaner.threads`, and so on.

- Typically, we would want to set `num.io.threads` to greater than the number of data disks since data will be hitting the page cache as well as disk.
- `num.network.threads` should be doubled if using TLS
- It's common to put extra threads into `num.replica.fetchers`
- `log.cleaner.threads` can be set up to the number of disks or the number of cores depending on whether log cleaning is I/O bound or CPU bound (it is usually I/O bound)
  - In either case, it is good to also set `log.cleaner.io.max.bytes.per.second` (Default: unbounded) to throttle log cleaning if it is degrading Broker performance.



As mentioned in a previous module, TLS overhead can cause significant CPU overhead for Brokers running with Java 8. Overhead is significantly reduced in Java 11, which is supported as of CP 5.2. Also, Brokers are now able to communicate with zookeeper over TLS (v5.4CP) adding a little more cpu requirements.

# Capacity Planning: Number of Brokers

- Storage

```
Number of Brokers =  
(messages per day * message size * Retention days * Replication) / (disk space per  
Broker)
```

- Network bandwidth

```
Number of Brokers =  
(messages per sec * message size * Number of Consumers) / (Network bandwidth per  
Broker)
```



Maximums: 4,000 Partitions per Broker and 200,000 Partitions per cluster.

We can take the maximum of these estimates for number of Brokers. These calculations provide reasonable first estimates based on what the limited resource is in your environment. Use this to set initial cluster size, but be prepared to adjust this once you are using Kafka in production.

Whatever number is derived from these calculations should then be rounded up to account for:

- Future growth
- Traffic spikes
- Failover capacity (i.e., if you have 3 Brokers and 1 fails, the other 2 need to have enough capacity to compensate for the failed Broker)

Typically, very large Brokers can sustain a maximum of 2000-4000 Partitions; most installations will be in the 1000's range. As a rule of thumb, we recommend each Broker to have no more than 4,000 Partitions and each cluster to have no more than 200,000 Partitions. The main reason for the latter cluster-wide limit is to accommodate for the rare event of a hard failure of the controller (i.e., a crash of the Broker system, rather than a software failure). This means the minimum deployment of a cluster with 200,000 Partitions is 50 Brokers of 4,000 Partitions each.

# Deploying Kafka in the Cloud

- Self-managed cloud deployment:
  - Memory optimized compute instances
  - Multiple availability zones (`broker.rack`)
  - Private subnet for inter-Broker traffic
  - Private subnet for ZooKeeper
  - Lockdown firewall rules, Kafka security
  - For AWS: "EBS optimized" instances



Virtual cores are weaker than physical cores

- Or consider:



Confluent  
Cloud

Leverage Kafka's rack awareness feature by assigning `broker.rack` value according to availability zone. More specific guidelines about placement of Brokers and ZooKeeper nodes across AZs will be discussed in the multi-datacenter section of this chapter.

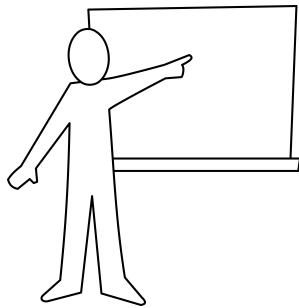
Distributed storage is discouraged for Kafka on-premise deployments because of the importance of disk I/O, but "EBS optimized" instances on AWS have proven to be stable and performant. AWS Elastic Block Store (EBS) is a distributed storage system that is automatically replicated within its availability zone to protect from hardware failure. The EBS tier chosen could be SSDs or throughput optimized HDDs (`st1`), depending on

price/performance needs. As aforementioned, HDDs tend to be a good choice for Brokers because of the heavy use of the in-memory page cache. Keep in mind that entire availability zones can fail, so Kafka's own replication mechanism is still incredibly important to prevent data loss and maintain high availability.

Confluent offers a managed cloud product that incorporates the entire reference architecture studied in this module, not just Apache Kafka.

tejaswin.renugunta@walgreens.com

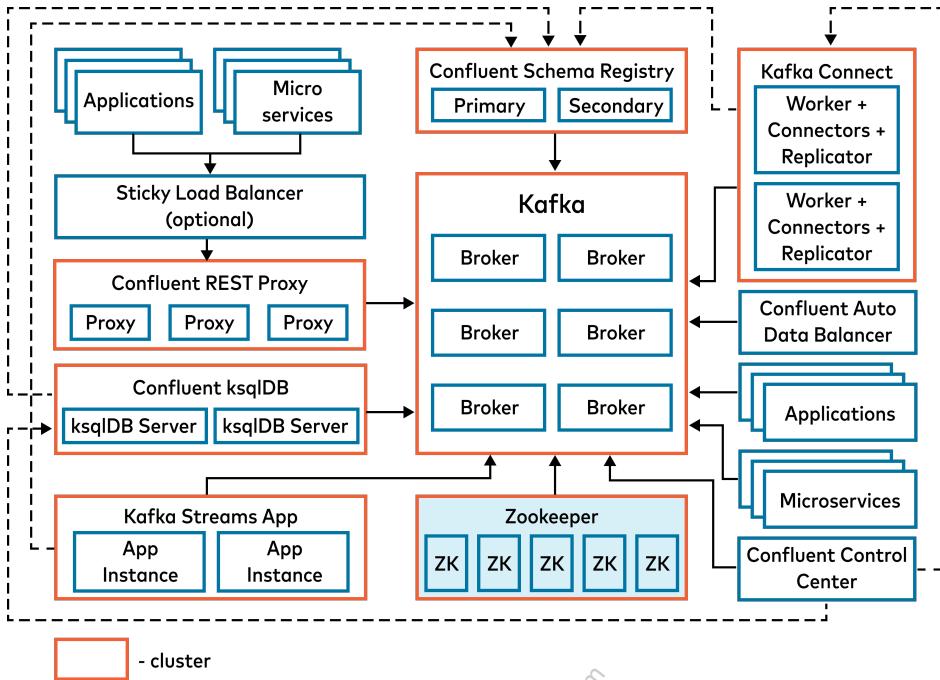
# Module Map



- Typical Kafka Design Model
- Brokers
- ZooKeeper ... ↙
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers

tejaswin.renugunta@walgreens.com

# Kafka Reference Architecture: ZooKeeper



ZooKeeper is a centralized service for managing cluster metadata and is a mandatory component in every Apache Kafka cluster. While the Kafka community has been working to reduce the dependency of Kafka clients on ZooKeeper, Kafka Brokers still use ZooKeeper to manage cluster membership and elect a cluster Controller. In order to provide high availability, you will need at least 3 ZooKeeper nodes (allowing for one-node failure) or 5 nodes (allowing for two-node failures). All ZooKeeper nodes are equivalent, so they will usually run on identical nodes. The number of ZooKeeper nodes **must be odd**.

# Capacity Planning: ZooKeeper

- Must be **odd** number of ZooKeeper instances:
  - ZooKeeper works by Quorum, i.e., majority votes
  - 3 nodes allow for one node failure
  - 5 nodes allow for two node failures (recommended)
  - Deploy across **availability zones**
- Hardware:
  - RAM: 8 GB
  - Transaction log (`dataLogDir`): 512 GB SSD
  - Database snapshots (`dataDir`): 2 TB RAID 10
  - CPU: Minimal 2-4 cores

---

If a ZooKeeper ensemble has so many failures that it does not have a quorum of nodes (i.e. more than half the starting number of nodes), the ensemble cannot function. If ZooKeeper fails, so do the Kafka Brokers which it supports. ZooKeepers consider a message committed after a quorum has written to disk. For this reason, SSDs are important.

Most of the traffic between ZooKeeper and the Brokers is mission-critical but low-volume: updating Topic metadata, Broker registration and status updates, and ACL storage and lookup. This data has to be low-latency so that the cluster is not slowed down by these lookups.

For ZooKeeper deployments in cloud environments, ZooKeeper nodes should be in different AZs to prevent a single AZ failure from affecting more than one node. For example, imagine a deployment where AZ1 has one ZooKeeper node and AZ2 has two ZooKeeper nodes. If AZ1 fails, two ZooKeeper nodes exist and the cluster continues to run since there is a quorum. However, if AZ2 fails, only a single node remains in AZ1 and ZooKeeper is effectively disabled. A better solution would be to put each ZooKeeper node in a separate AZ since the chances of losing two AZs simultaneously are reasonably low.

# Disk Space on ZooKeeper

- ZooKeeper saves **snapshots** and **transactional log** files
  - Transaction log should have a dedicated SSD
  - Configure ZooKeeper to purge snapshots in its `server.properties` file:
    - `autopurge.snapRetainCount`: number of snapshots to retain
    - `autopurge.purgeInterval`: hours between purges
- 

Snapshots are persistent copies of znodes (znodes are how ZooKeeper stores data—essentially directories that also hold data, or files that also have children).

The transaction log is a log of events that occur in ZooKeeper (much the same as a log in Kafka). ZooKeeper builds and maintains its state store from the transaction log. As shown in the previous slide, it is important for the transaction log to have its own dedicated storage.

As with most storage-based services, running out of disk space is not a good idea. Leverage ZooKeeper's built-in tools to manage the disk-space usage. Confluent documentation suggests `autopurge.snapRetainCount=3` and `autopurge.purgeInterval=24`, which purges all but 3 snapshots every 24 hours. If these settings are used, it may be feasible to use less storage than mentioned on the previous slide, but then again for such a mission critical system, a little extra storage on 3-5 nodes is reasonable.



Snapshot files can grow to be large.

# Monitoring ZooKeeper

- ZooKeepers have their own JMX metrics:
- **NumAliveConnections** - make sure you are not close to maximum as set with **maxClientCnxns** (Default: 60)
- **OutstandingRequests** - should be below 10 in general
- **AvgRequestLatency** - target below 10 ms
- **MaxRequestLatency** - target below 20 ms
- **HeapMemoryUsage** (Java built-in) - should be relatively flat and well below max heap size



Use `nc` or `telnet` to send ZooKeeper "four letter word" commands: `mntr` for basic monitoring or `ruok` for basic health check

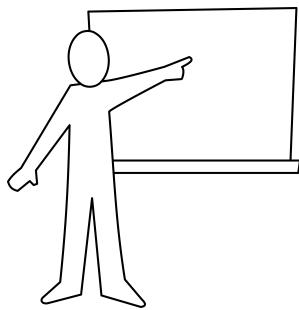
ZooKeeper is critical; if it fails, so do the Brokers. Kafka has JMX metrics it uses to measure the health of its connection to ZooKeeper, and ZooKeeper itself has its own JMX metrics.

A simple health check is the command `ruok`. ZooKeeper will respond with `imok` if it is ok. To monitor some of these metrics easily, the `mntr` command will monitor everything listed except heap usage and number of alive connections. ZooKeeper has several of these "four letter word" commands that can be issued over TCP with netcat:

```
echo "<command>" | nc <host> <port>
```

For more details about ZooKeeper's "four letter word" commands, see  
[https://zookeeper.apache.org/doc/r3.4.8/zookeeperAdmin.html#sc\\_zkCommands](https://zookeeper.apache.org/doc/r3.4.8/zookeeperAdmin.html#sc_zkCommands).

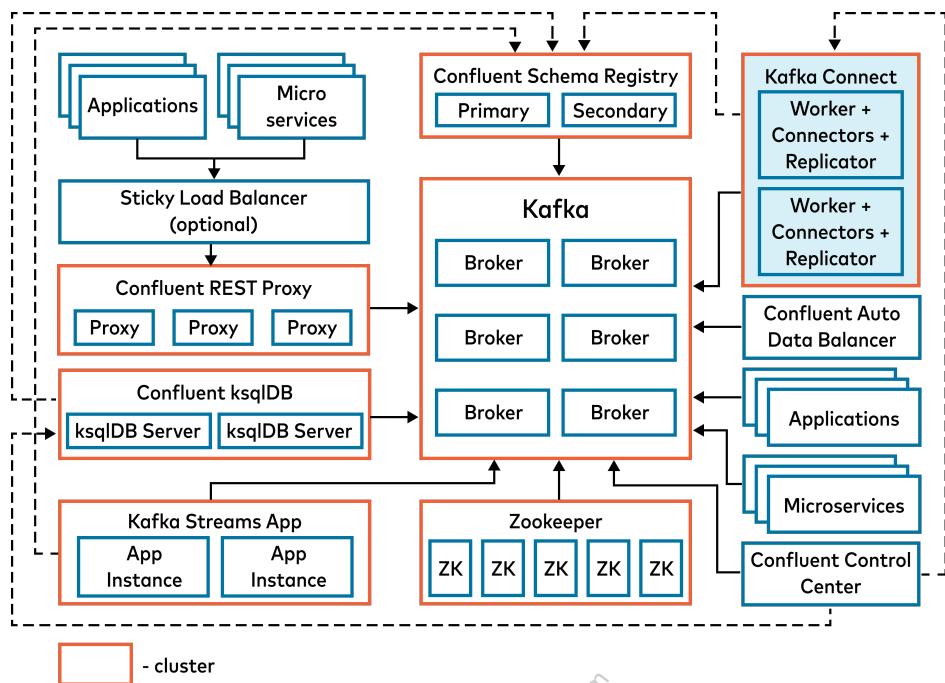
# Module Map



- Typical Kafka Design Model
- Brokers
- ZooKeeper
- Kafka Connect ... ←
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers

tejaswin.renugunta@walgreens.com

# Reference Architecture: Kafka Connect



tejaswin.renugunta@walgreens.com

# Connect Workers for High Availability

- Deploy machines with same `group.id` to form cluster
- Deploy at least 2 machines behind load balancer
- Add machines with same `group.id` to add capacity

---

Kafka Connect workers should not be run on Brokers. They are deployed on servers separate from the Kafka cluster itself.



Cooperative rebalancing in Kafka 2.3 improves Connect cluster rebalances since there is less pausing during rebalance. See:  
<https://cwiki.apache.org/confluence/display/KAFKA/KIP-415%3A+Incremental+Cooperative+Rebalancing+in+Kafka+Connect>.

tejaswin.renugunta@walgreens.com

# Tune `tasks.max` for Scalability

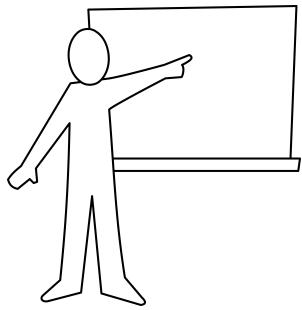
- `tasks.max` is a config sent to Connect via REST API
  - Set to minimum of:
    - Number Topic-Partitions (for sink connectors)
    - Desired throughput / Throughput per task
    - Machines \* Number of cores per machine
  - The more cores in the Connect cluster, the better
- 

Connect workers are designed to be multi-threaded and so will benefit from multi-core servers.

The more cores, the more threads can run simultaneously. You can configure the worker to take advantage of the multi-core environment by tuning the `tasks.max` property. This property is specified in the REST `http` request. For example:

```
curl -X POST -H "Content-Type: application/json" --data \  
'{  
    "name": "local-file-sink",  
    "config": {  
        "connector.class": "FileStreamSinkConnector",  
        "tasks.max": "1",  
        "file": "test.sink.txt",  
        "topics": "connect-test"  
    }  
' http://connect-1:8083/connectors
```

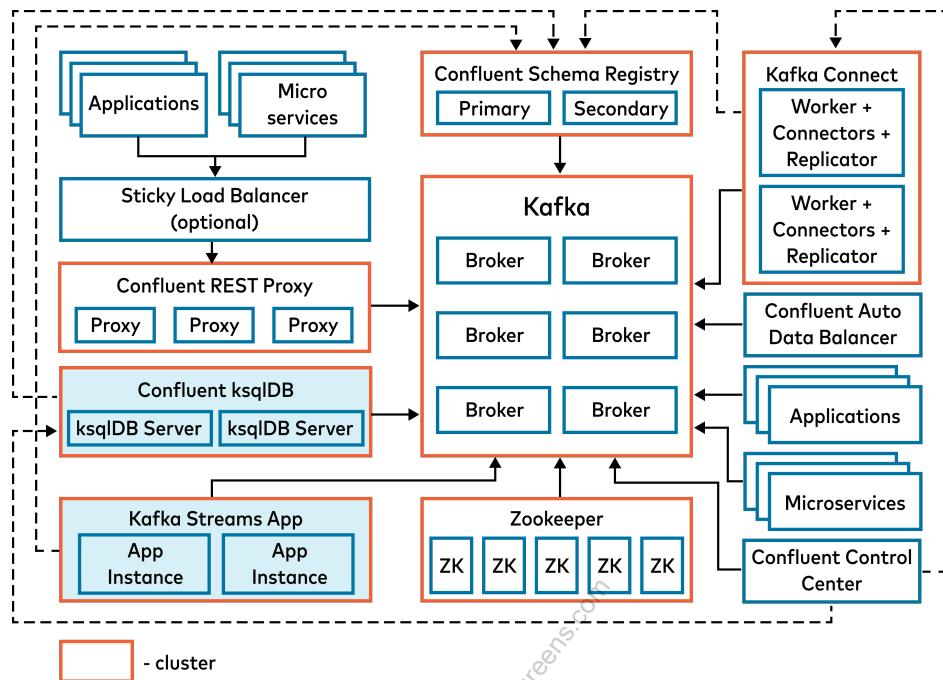
# Module Map



- Typical Kafka Design Model
- Brokers
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB ... ←
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers

tejaswin.renugunta@walgreens.com

# Kafka Reference Architecture: Kafka Streams and ksqlDB



On the next slide, there is a brief overview of the Kafka Streams API and related deployment considerations.

# Kafka Streams and ksqlDB Applications

Deploy multiple instances,  
multiple clusters

- Load balancing via Kafka
- Failover via Kafka
- One cluster per team or app

Configure `num.standby.replicas`

- Pro: Faster task failover
- Con: More load on Kafka cluster



Consider using Kafka quotas to apply a throttling mechanism to ensure high performance for all clients

At its core, **Kafka Streams** API is a Java library that can be used inside of a company's business applications. Applications running with the Kafka Streams API will benefit from being run on many servers, each with many cores, all identified together by the `application.id` configuration setting in the application code. The library itself handles load balancing and failover of tasks across instances. Each instance of the application maintains a local state store in RocksDB. **ksqldb** is a technology that creates Kafka Streams applications via a SQL-like interface, making many stateful stream processing tasks much easier and more user-friendly. ksqldb servers are identified together by the `ksql.service.id` property.

Standby replicas are shadow copies of local state stores. Kafka Streams attempts to create the specified number of replicas per store and keep them up to date as long as there are enough instances running. Standby replicas are used to minimize the latency of task failover. A task that was previously running on a failed instance is preferred to restart on an instance that has standby replicas so that the local state store restoration process from its changelog can be minimized. If there are no standby replicas, then the state store is rebuilt from data persisted in Kafka itself.

Kafka Streams applications read in from and write out to Topics. This will increase the number of Topics per Cluster in many cases, requiring administrators to plan for more Broker capacity. Standby replicas put added pressure on the cluster as well since they are redundant Consumers that take up network bandwidth.

# Kafka Streams and ksqlDB Servers

## Capacity Planning:

- 100-300 GB SSD for local RocksDB state store
- 4+ CPU Cores
- 32+ GB RAM
- 1 Gbit network

## ksqlDB:

- Interactive mode
- **Headless mode** (recommended for production)

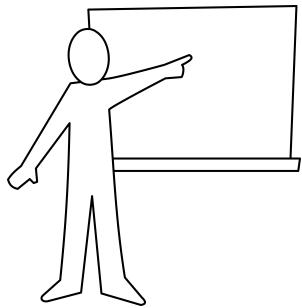
---

When deploying a ksqlDB cluster, there are 2 possible modes in which the servers can run: interactive and headless mode. Interactive mode allows access to ksqlDB's REST API and command line interface, whereas headless mode runs queries from a `.sql` file. In principle, it is recommended that production environments run in headless mode for planning and security reasons; however, there are cases where interactive mode is the only option. This could be the case when you have Kafka Client applications that are written in any non Java language and are accessing functionality of the ksqlDB cluster via its REST API.

## What do we have to consider?

- Headless mode: The queries running on the cluster are well known in advance and have been extensively tested. We have a good understanding of how many cluster nodes we need based on our calculations and performance tests. For more information about deploying in headless mode, see <https://docs.confluent.io/current/ksql/docs/installation/server-config/index.html#non-interactive-headless-ksql-usage>
- Interactive mode: Here the situation is not so easy. The moment access is granted to the ksqlDB Server REST API, it is not always clear how many queries are going to run on my cluster at any given time. The whole sizing of the cluster is harder to do correctly.

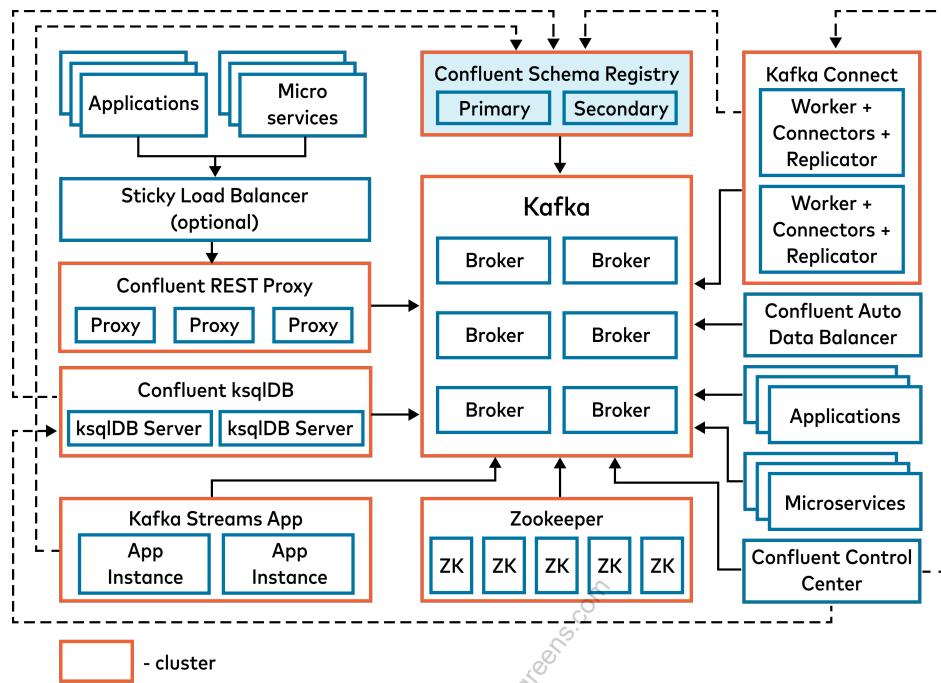
# Module Map



- Typical Kafka Design Model
- Brokers
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry ... ↙
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers

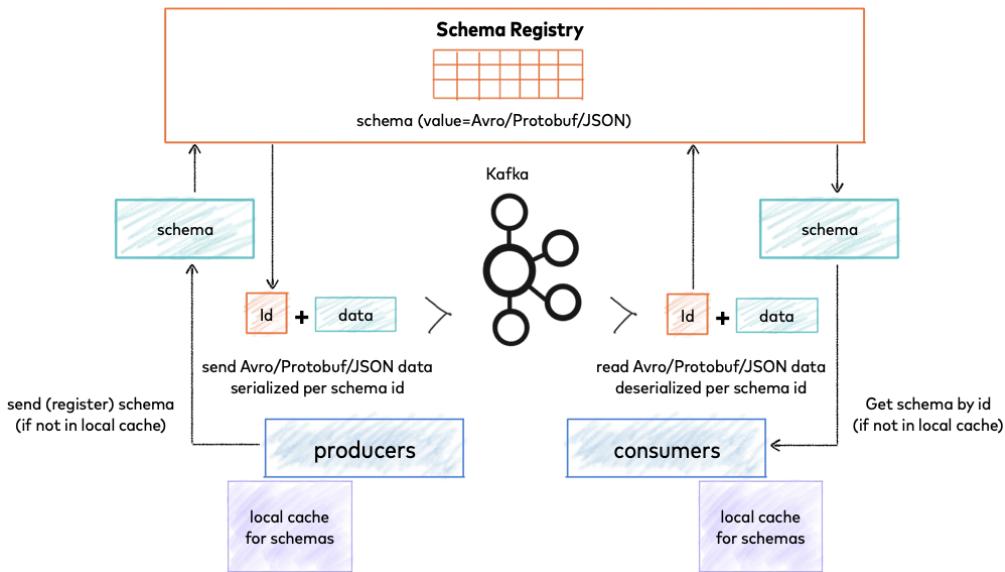
tejaswin.renugunta@walgreens.com

# Reference Architecture: Confluent Schema Registry



Notice the primary/secondary architecture of Schema Registry. There can be one primary node that accepts reads and writes, and many read-only secondary nodes.

# Schema Registry Overview



As business changes or more applications want to leverage the same data, there is a need for the existing data structures to evolve. We need what's called a schema evolution. Poor planning can mean that each schema change could potentially break an application that wasn't aware something changed.

The Confluent Schema Registry manages changing schemas so that those changes don't break Consumers. Messages are serialized using a serializer and the schema is sent via a REST API to Schema Registry with a schema ID. There is a special single-Partition Kafka Topic (`_schemas`) where schema information persists and is replicated. This Topic has the "compact" retention policy. Schema Registry machines also cache schema data locally.

The schemas assigned to the key and value of the messages in a Topic can be viewed by examining the Topic in Confluent Control Center. The interface also shows the version history if the schema has changed over time.

Confluent Platform 5.5 now supports **Protocol Buffers**, **JSON** and **Avro**, the original default format for Confluent Platform. Support for these new serialization formats is not limited to Schema Registry, but provided throughout Confluent Platform. Additionally, as of Confluent Platform 5.5, Schema Registry is extensible to support adding custom schema formats as schema plugins.

The Schema Registry is covered in more detail in the Dev course and in these references:

- <https://docs.confluent.io/current/avro.html>

- <https://docs.confluent.io/current/schema-registry/docs/operations.html#>



If the `kafkastore.topic.replication.factor` (Default: 3) property in the Schema Registry's properties file is greater than the number of Brokers, `_schemas`, Topic creation will still succeed. This is different from the behavior in Kafka 0.11.0 (Confluent 3.3) for the `--consumer_offsets` Topic, whereby `offsets.topic.replication.factor` will be enforced upon auto Topic creation.

tejaswin.renugunta@walgreens.com

# Capacity Planning: Schema Registry

- Minimal system requirements
- Deploy 2+ servers behind load balancer
- Single-primary architecture
  - One primary node at a time
  - Primary node responds to write requests
  - Secondary nodes forward write requests to primary node
  - All nodes respond to read requests

---

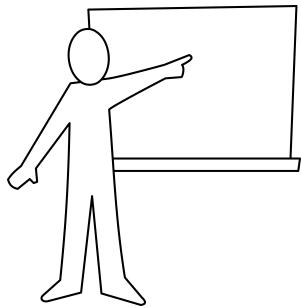
The Schema Registry is just a very simple lookup service so it does not require many resources. For example, it just needs a 1 GB JVM heap. State is stored in Kafka, so there are minimal storage requirements. There is little load on the CPU.

Schema Registry is mission-critical once deployed, so it should be deployed as a cluster for high-availability. The Schema Registry cluster should be placed behind a load balancer for ease of configuration.

The cluster works in a primary/secondary setup. Only the primary node can write to the `_schemas` Topic in Kafka. Secondary nodes can forward write requests to the primary node. Primary election is handled by Kafka if `kafkastore.bootstrap.servers` is set to a comma-separated string of Kafka endpoints, or by ZooKeeper if `kafkastore.connection.url` is set instead. If both are set, ZooKeeper will handle primary election. Kafka-based primary election was introduced in CP 4.0., and is discussed in detail in this talk: <https://youtu.be/MmLezWRI3Ys?t=1776>.

	<p>It is recommended to set <code>min.insync.replicas</code> to 2 or greater and <code>unclean.leader.election.enable=false</code> on the <code>_schemas</code> Topic from the Kafka cluster. These configurations are not set from Schema Registry itself.</p>
---	---

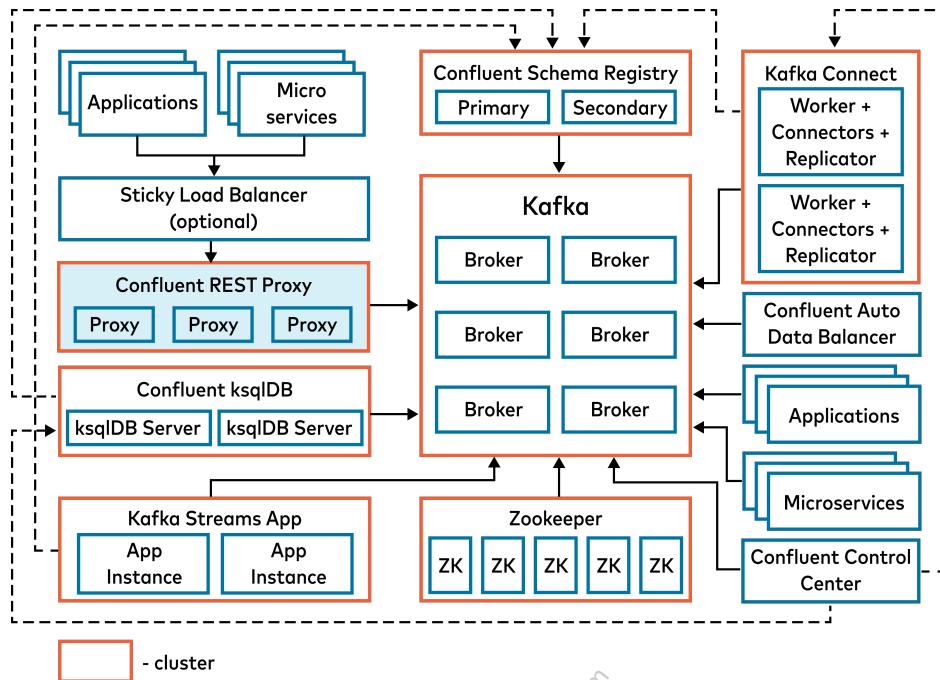
# Module Map



- Typical Kafka Design Model
- Brokers
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy ... ←
- Confluent Control Center
- Multiple Data Centers

tejaswin.renugunta@walgreens.com

# Reference Architecture: Confluent REST Proxy



The Kafka REST Proxy is a part of Confluent Community. It provides a way to have Producer and Consumer style interaction with the Brokers through HTTP calls. This is beneficial for:

- Companies that use languages that don't have first-class Kafka client library support
- Prototyping application logic without worrying about the Java API
- Easily viewing cluster information or other administrative actions

Any language that can make HTTP requests can now be used to write Producers and Consumers!

Following the pattern of other components, REST Proxy configuration settings are located at </etc/kafka-rest/kafka-rest.properties>. For more information, see <http://docs.confluent.io/current/kafka-rest/docs/config.html>.

# Capacity Planning: REST Proxy

- Memory: Buffers both Producers and Consumers
    - $1000 \text{ MB RAM} + (P * 64\text{MB}) + (C * 16 \text{ MB})$
  - 16+ CPU cores
  - "Sticky load balancer" needed for Consumers
  - Stateless → Container orchestration!
- 

REST Proxy buffers data for both Producers and Consumers. Consumers use at least 2MB per Consumer and up to 64MB in cases of large responses from Brokers (typical for bursty traffic). Producers will have a buffer of 64MB each. Start by allocating 1GB RAM and add 64MB for each Producer and 16MB for each Consumer planned.

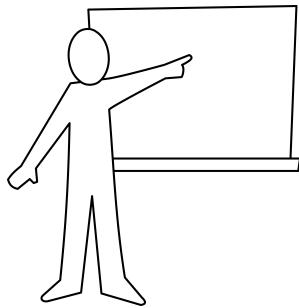
We recommend at least 16 cores, which provides sufficient resources to handle HTTP requests in parallel and background threads for the Producers and Consumers. However, this should be adjusted for your workload. Low throughput deployments may use fewer cores, while a proxy that runs many Consumers should use more because each Consumer has a dedicated thread.

The REST Proxy is typically deployed as a cluster for performance and high-availability. If using a load balancer, make sure to enable the "sticky session" feature (also known as "session affinity"). This feature enables the load balancer to bind a user's session to a specific instance. This ensures that all requests from the user during the session are sent to the same instance. This is required for Consumers.



From a security perspective, REST Proxy collapses the identities of all clients to a single identity. By default, all the requests to the Broker use the same Kerberos Principal or the SSL certificate to communicate with the Broker when the `client.security.protocol` is configured to be either of `SSL`, `SASL_PLAIN`, or `SASL_SSL`. Confluent Enterprise Security Plugins are required to set fine-grained ACLs for individual clients. For more information, see <https://docs.confluent.io/current/kafka-rest/security.html>.

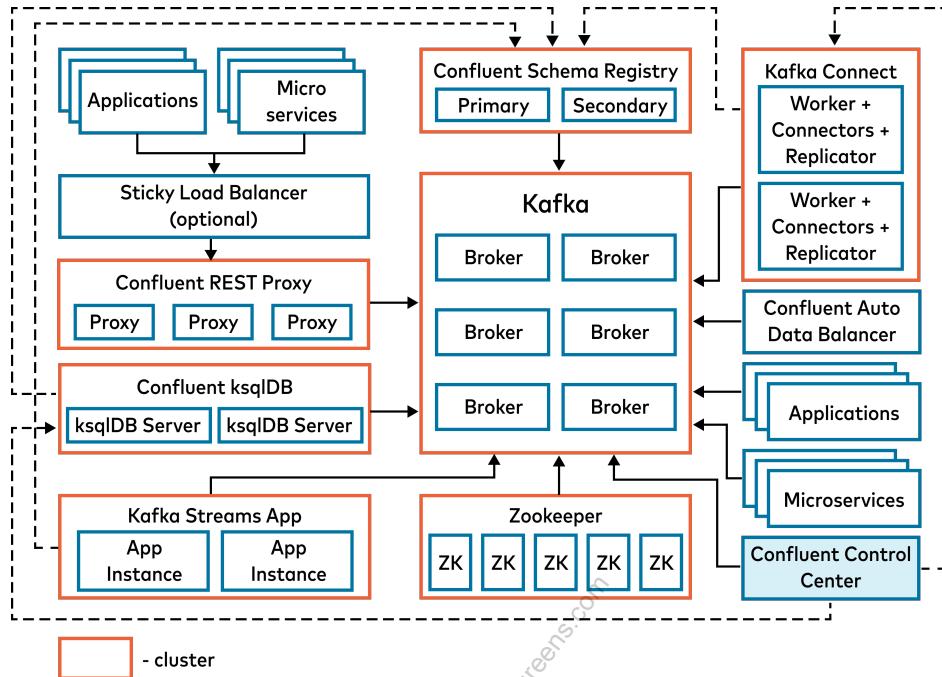
# Module Map



- Typical Kafka Design Model
- Brokers
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center ... ←
- Multiple Data Centers

tejaswin.renugunta@walgreens.com

# Reference Architecture: Confluent Control Center



Each component in the architecture can be equipped with monitoring interceptors in its configuration to publish metrics to a Kafka cluster. Confluent Control Center connects to a primary Kafka cluster that holds all of these metrics.



Remember that CCC requires an enterprise license.

# Highly Available Confluent Control Center

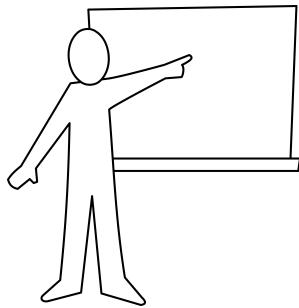
The screenshot shows the CCC interface. At the top, there's a banner for an 'Enterprise trial ends in 28 days'. Below that, the 'Home' section has a green box for 'Healthy clusters' (1) and a red box for 'Unhealthy clusters' (0). There's a search bar and a 'Hide healthy clusters' toggle. A main card for 'controlcenter.cluster' is expanded, showing detailed metrics: 3 Brokers, 246 Partitions, 44 Topics, 33.5kB/s Production, and 26.7kB/s Consumption. It also lists 'Connected services' with 1 KSQL cluster and 1 Connect cluster.

- Deploy a **separate Kafka cluster** dedicated for metrics
- Deploy 2+ machines and load balancer dedicated to CCC
- Configure each UI server with a unique `confluent.controlcenter.id`

In the reference architecture diagram, it shows a single CCC instance connected to a Kafka cluster that is also used to handle other business logic. This may be acceptable in early stages of a company's infrastructure, but best practices dictate that metrics analysis should be separate from the system it is analyzing. That means there should be a Kafka cluster dedicated to metrics. For the UI itself to be highly available, there should be multiple UI servers deployed behind a load balancer with virtual IP.

Like other components in the Kafka + Confluent ecosystem, CCC has a `/etc/confluent-control-center/control-center.properties` file that can be used to configure connection to a dedicated metrics Kafka cluster, metrics retention, and connections to named Kafka clusters and components (e.g. `confluent.controlcenter.kafka.production-nyc.bootstrap.servers` or `confluent.controlcenter.connect.production-nyc.cluster`).

# Module Map

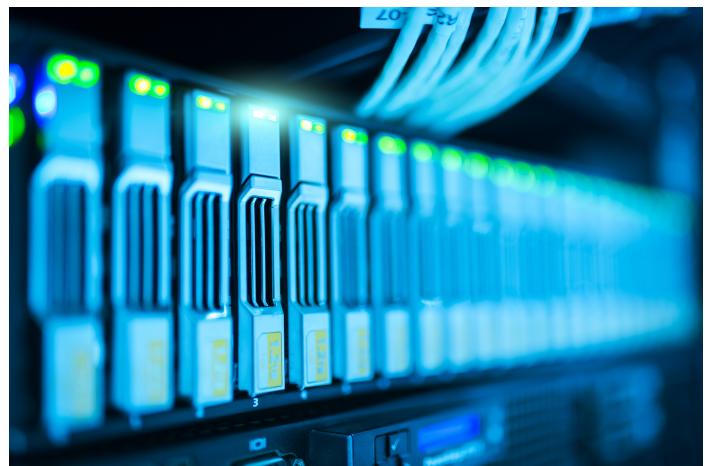


- Typical Kafka Design Model
- Brokers
- ZooKeeper
- Kafka Connect
- Kafka Streams and ksqlDB
- Confluent Schema Registry
- Confluent REST Proxy
- Confluent Control Center
- Multiple Data Centers ... ←

tejaswin.renugunta@walgreens.com

# Multiple Data Centers

- Kafka only:
  - Stretched (a.k.a. multi-AZ)
- Confluent Replicator:
  - Cluster aggregation
  - Active/Passive
  - Active/Active



---

There are many use cases for multiple datacenter deployments of Kafka clusters. Using multiple DC's, it is possible to design for DC-wide failure scenarios. If many Kafka clients connect to a Kafka cluster in a different DC, it would also save cross-DC bandwidth to replicate cluster data and then configure those clients to interact with the cluster in the local DC instead.

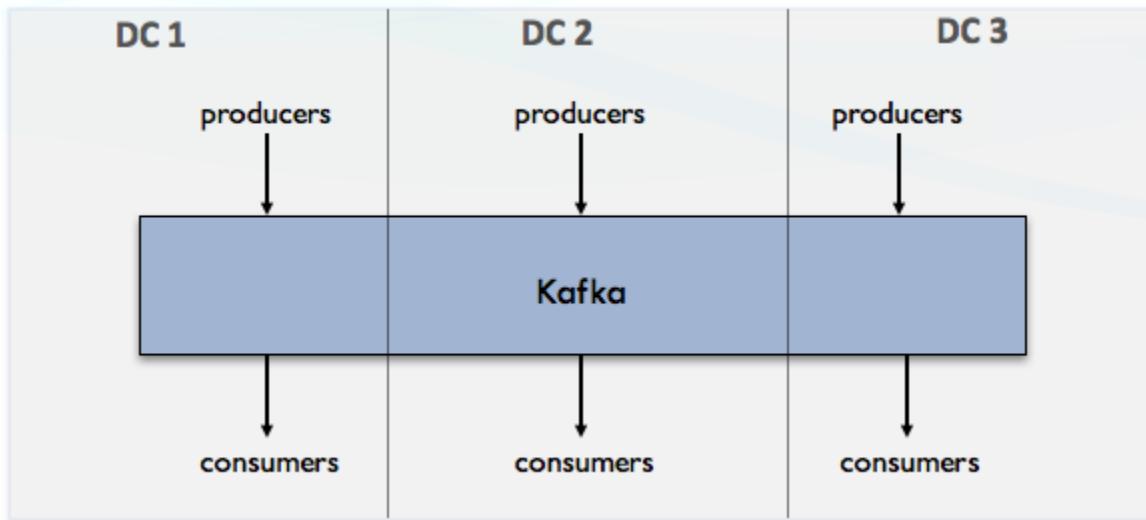
This section is based off of the white-paper available from:

<https://www.confluent.io/white-paper/disaster-recovery-for-multi-datacenter-apache-kafka-deployments/>

Refer to this site for a discussion of data aggregation:

<https://www.confluent.io/blog/enterprise-streaming-multi-datacenter-replication-apache-kafka/>

# Stretched Deployment



- Discussion Questions
  - How should you deploy ZooKeeper and Kafka across 3 availability zones?
  - What are possible tradeoffs between a stretched cluster vs. a single DC cluster?
  - What are some possible failure scenarios and how does Kafka respond?

A "Stretched" deployment is a single logical cluster across DCs that are close to each other. An example of this would be availability zones in AWS or GCP that are within the same region. This has already been discussed in a previous module, but is presented again here to reiterate that Kafka's built-in replication mechanisms make it well-suited for a multi-AZ deployment.

Here are some ideas to consider for discussion if not mentioned by students:

- How should you deploy ZooKeeper and Kafka across multiple availability zones?
  - ZooKeeper nodes should be placed in such a way that the failure of a single zone will not disrupt quorum.
  - Brokers should be configured with `broker.rack=<zone-name>` so that Partition replicas are spread across zones.
  - Brokers should be evenly spread amongst the availability zones.

- What are possible tradeoffs between a stretched cluster vs. a single DC cluster?
  - A stretched cluster will have increased latency.
  - Stretched cluster is more highly available in the case that an entire availability zone goes down.
  - Disruption in networking could cause replication to fail and time-based features (e.g., Brokers checking in with ZooKeeper, Followers timing out of the ISR list) to fail.
- What are some possible failure scenarios and how does Kafka respond?
  - If an entire AZ goes down, Kafka's replication mechanisms will work like usual to keep the cluster available. Consumers at the other AZ's will resume from last committed offset.
  - If two AZ's fail, then ZooKeeper will lose quorum and thus the Kafka cluster will be down.
  - In very rare cases, it's possible for Brokers in one AZ to lose connection to Brokers in other AZ's while *retaining* connection to the ZooKeeper ensemble. In this case, Partition leaders in the isolated AZ will drop followers in the other AZ's from the ISR list and vice versa. This will lead to under-replication, although the cluster will still be available. If a leader in the isolated AZ fails and the Controller is in a different AZ, then new leader election might fail if the Controller isn't able to send the metadata update to the new leader in the isolated AZ. This will be mitigated with rack awareness since it is likely that there will be a candidate for leader election in a different AZ that the Controller can reach.

tejaswin.renugopal@engg.co

# Cross-DC Replication

## Confluent Replicator or Apache Kafka MirrorMaker

For cross-regional replication, latency can be too great to run a stretched Kafka cluster. In this case, it is recommended to use a replication technology like Apache MirrorMaker or Confluent Replicator to implement active/active, active/passive, or aggregation Kafka clusters.

Here are some facts about Confluent Replicator at a glance:

- Kafka Connect source connector
- Consumer offsets preserved via offset translation
- Replicates **data and metadata**
- Recommended to deploy in dedicated Kafka Connect cluster
- Requires Confluent Enterprise License

**MirrorMaker:** MirrorMaker is an open source technology used for cross data center replication that comes as a part of core Apache Kafka. For the most up to date information about MirrorMaker, see

[https://kafka.apache.org/documentation/#basic\\_ops\\_mirror\\_maker](https://kafka.apache.org/documentation/#basic_ops_mirror_maker).

**Replicator:** Confluent Replicator is a proprietary Kafka Connect source connector, which means it inherits all the benefits of the Kafka Connect API including scalability, performance, and fault tolerance. A major benefit of Replicator is that it will configure the destination Topic to match the structure (e.g., Partition count, replication factor) of the source Topic. Since Replicator is designed to run continuously, it will also pass configuration changes (e.g., retention time) automatically. It is recommended to deploy Replicator in its own dedicated Connect cluster so that it can be tuned specifically for cross-DC replication and so other connectors don't interfere with its operation. Remember that this is accomplished by configuring the machines with a `group.id` dedicated for Replicator, e.g. `group.id=dc1-to-dc2-replicator`.

Question: Should Replicator be deployed in the source DC, or the destination DC? Why?

- The Replicator connector functions as both a Producer (as it reads from the source Topic) and Consumer (as it writes to the destination Topic). Replicator should be closer to destination site because Producers are more sensitive to network interruptions than

Consumers. If a Consumer doesn't get a message, it will just retry; if a Producer request fails, there's the possibility that an acknowledged request will be resent or lost, depending on the `acks` setting.



As of CP 4.1, Replicator does not require a direct connection to ZK. Any communications with ZK will be passed through the Brokers.

tejaswin.renugunta@walgreens.com

# Deploying Replicator

1. Provision machines in **destination data center**
2. Install with `confluent-hub` if not already using CP
3. Configure `worker.properties` file on each Connect machine
4. Ways to start Replicator:
  - Submit HTTP request with Replicator-specific properties, **or**
  - Use the `replicator` command on each Kafka Connect machine



Confluent Replicator requires an enterprise license

---

Replicator is included in Confluent Platform, but requires the purchase of an enterprise license. If Kafka Connect machines weren't deployed with CP, then Replicator can be installed via the `confluent-hub` CLI, e.g.

```
$ confluent-hub install confluentinc/kafka-connect-replicator:latest
```

Like any other Connect cluster, specify a `worker.properties` file with a common `group.id` and other properties on each worker node and start the Connect JVM with the command:

```
$ connect-distributed worker.properties
```

Replicator can then be invoked on a Connect cluster via HTTP like any distributed connector.

Here is an example of calling Replicator via the REST API:

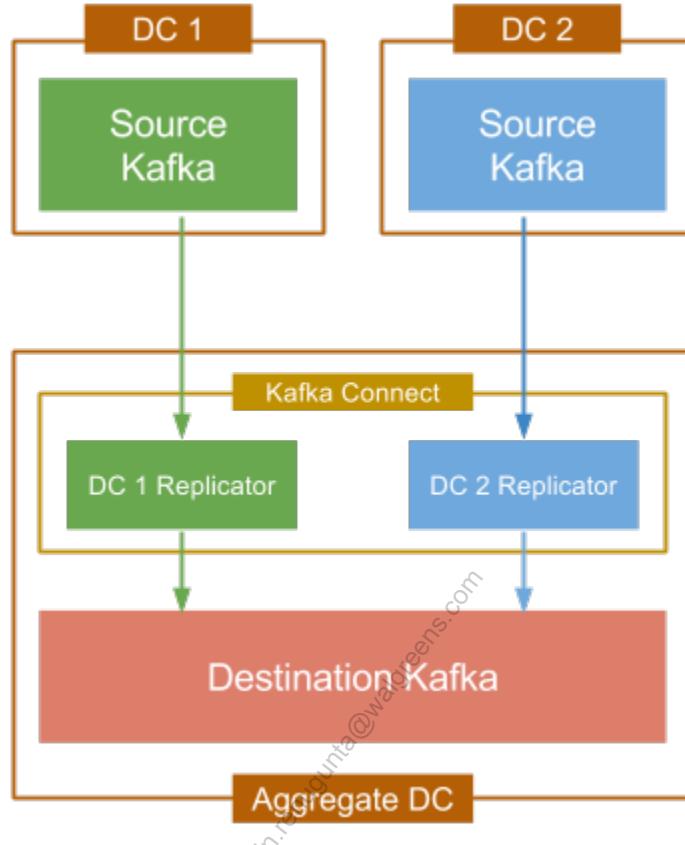
```
$ curl -s -X POST \
-H "Content-Type: application/json" \
--data '{
  "name": "dc1-to-dc2-replicator",
  "config": {
    "connector.class": "io.confluent.connect.replicator.ReplicatorSourceConnector",
    "tasks.max": 64,
    "provenance.header.enable": "true"
    "key.converter": "io.confluent.connect.replicator.util.ByteArrayConverter",
    "value.converter": "io.confluent.connect.replicator.util.ByteArrayConverter",
    "src.kafka.bootstrap.servers": "dc1-kafka-101:9092",
    "dest.kafka.bootstrap.servers": "dc2-kafka-101:9092"
    "producer.compression.type": "zstd"
    "producer.linger.ms": "1000"
    "producer.batch.size": "1000000"
    "consumer.max.partition.fetch.bytes": "10485760"
    "topic.whitelist": "test-topic",
    "topic.regex": "prod-.***"
    "topic.rename.format": "${topic}.replica",
    "confluent.license": "IAMALONGLICENSEKEY"
  }
}' http://dc2-connect:8083/connectors
```

There is also an executable called `replicator` included in CP that takes 3 property files (containing Producer, Consumer, and replication properties), and a `cluster.id` (which is equivalent to Connect's `group.id`) as arguments to start each Replicator worker:

```
$ replicator \
  --consumer.config ./consumer.properties \
  --producer.config ./producer.properties \
  --cluster.id dc1-to-dc2-replicator \
  --replication.config ./replication.properties
```

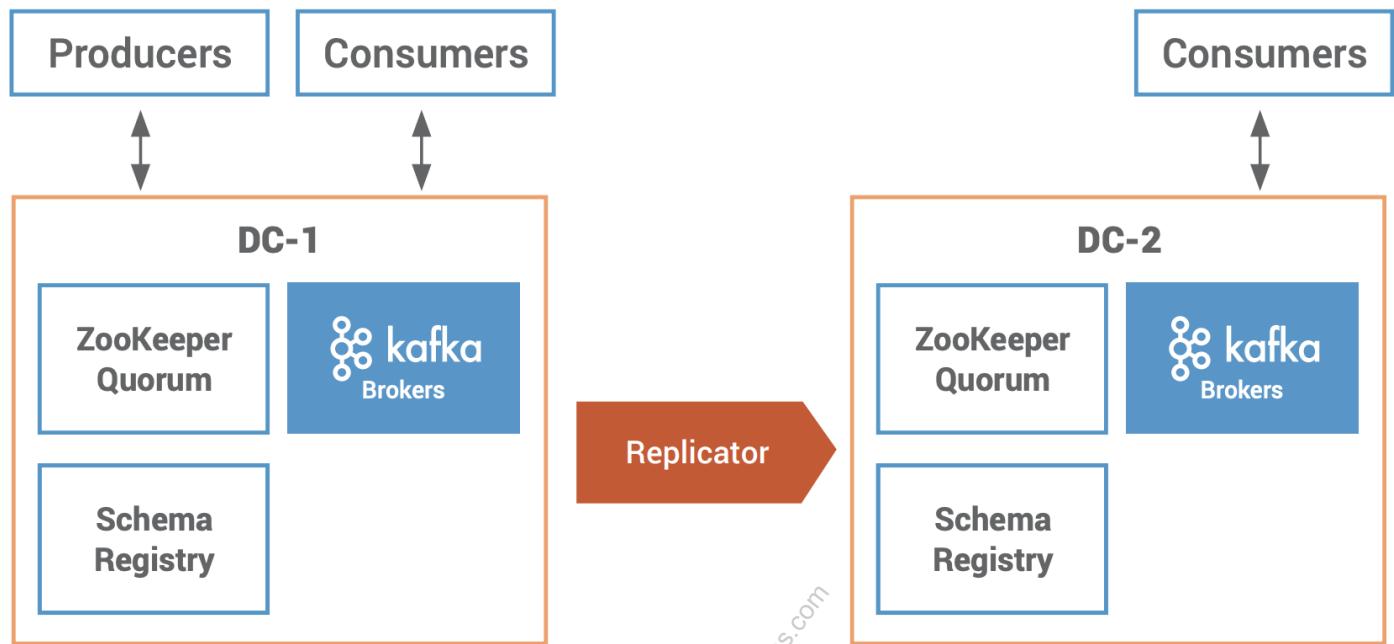
For more information on the replicator, see: <https://docs.confluent.io/current/multi-dc-deployments/replicator/index.html>.

# Cluster Aggregation



In this scenario, smaller, regional clusters are aggregated in a large central cluster. One interesting use case of this is a cruise ship business with local Kafka clusters on each cruise ship that connects to a centralized cluster upon return.

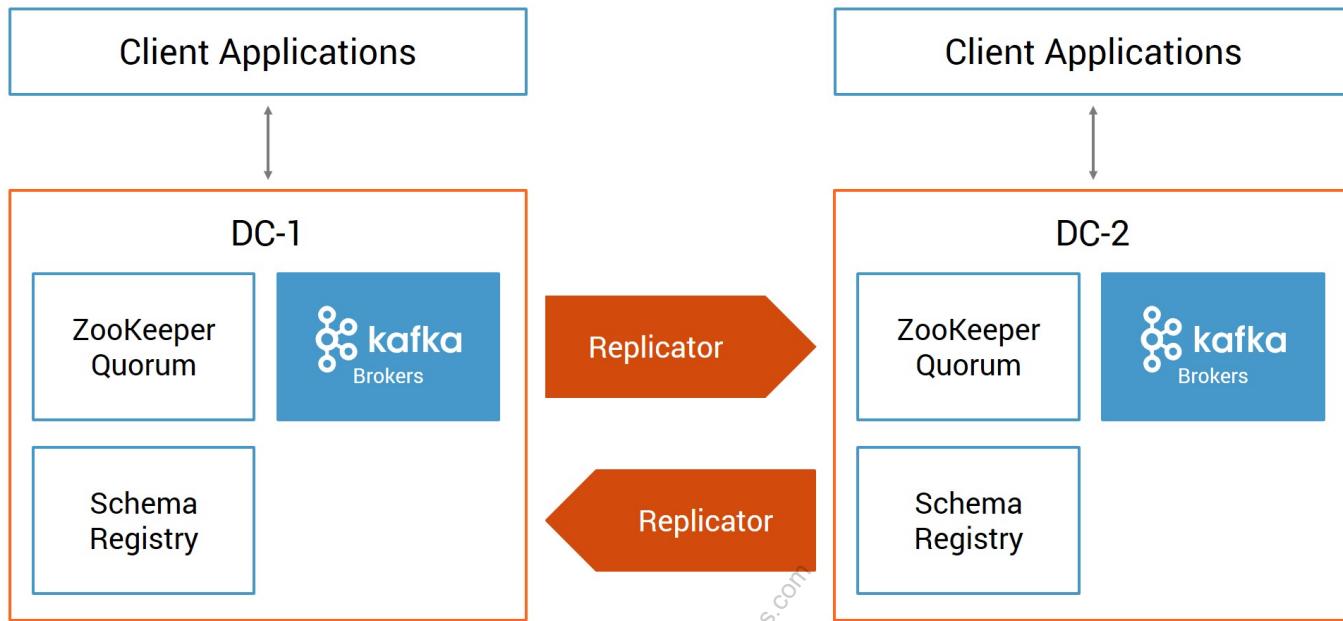
# Active/Passive



An "Active/Passive" deployment has two sites operating as independent clusters with their own ZooKeeper instances. Active/Passive environments are assumed to be asynchronously replicating. Since real-time performance is not expected, Replicator should be configured to use batching to maximize throughput. Overriding default client properties is accomplished by passing `consumer.<property>=value` and `producer.<property>=value` in the Connect REST API call.

Active/Passive is easier to implement than Active/Active. It is a good choice when you'd like to run geographically local Consumer applications and allow the failover of other Consumers, or would just like to back up the primary data center. As a downside to Active/Passive, the passive cluster may be underutilized.

# Active/Active

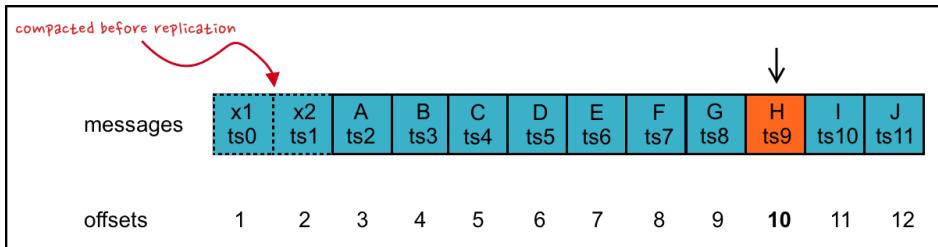


An "Active/Active" setup is where two DC's operate as independent clusters with their own ZooKeeper instances, but each also replicates its data to the other. This allows more flexible failover and better resource utilization. This is a good solution if you want client applications that are geographically local but where data is also shared more globally.

A common mistake with traditional Active/Active replication scenarios is the replication loop. If a Topic in one datacenter is replicating to a Topic with the same name in the other data center and vice versa, an infinite loop is created in which the same data is continuously passed between the two data centers. As of CP 5.0, Replicator uses the message header to record origin cluster information to prevent data being replicated back to the originating cluster (called "data provenance"). This feature will increase CPU utilization on the Replicator systems as it will be using CPU to filter all messages. To enable this feature, configure `provenance.header.enable=true`. This requires message format version 2.

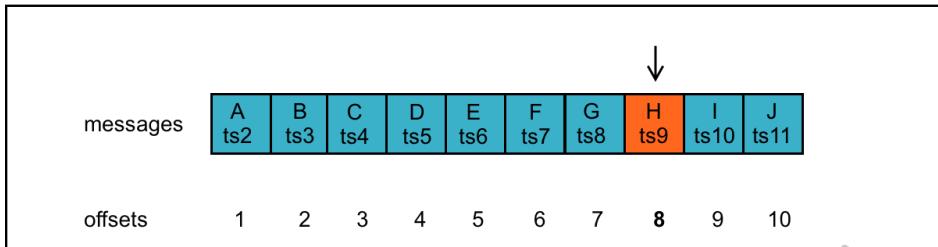
# Client Offset Translation (1)

Data Center 1



Consumer Group	Offset	Timestamp
my-group-1	10	ts9

Data Center 2



Consumer Group	Offset
my-group-1	8

Introduced in CP 5.0, Replicator enables a mapping of offsets to timestamps for messages in the replicated topics. This allows easier failover between sites.

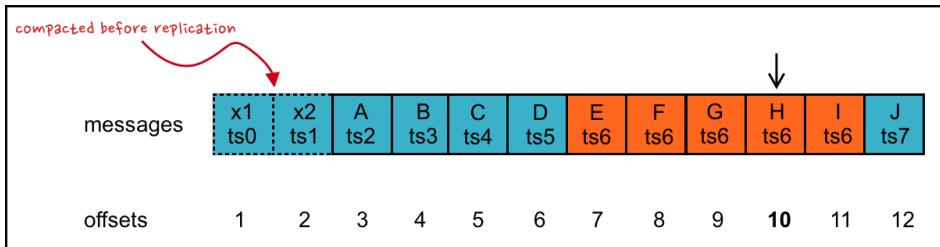
In order for this feature to work, the following requirements must be met in the Consumer:

- Add a jar dependency and add the following property to the Consumer:
  - `interceptor.classes=io.confluent.connect.replicator.offsets.ConsumerTimestampsInterceptor`
- Use message format v1 or later (includes timestamp)

Why doesn't Replicator preserve Consumer offsets by default (i.e., without the interceptors)? Using the same partitioner as the original Producer preserves the ordering (when using keys). But offsets may differ due to the sequential nature of offsets. Offset numbers must be written to in order, starting with 0; there is no way to start writing to a specific offset number. If a Topic that has existed long enough for some messages to age out, it no longer has access to offset 0. If that Topic is replicated, the newly created destination Topic must start numbering the offsets in the Partitions at 0, which would not match the source Topic offset numbers. Also illustrated in the slide is the case where logs are compacted before replication, which will also cause offsets in the destination cluster to differ from the source cluster.

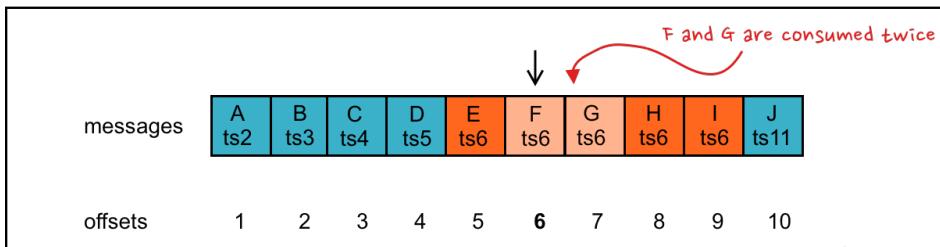
# Client Offset Translation (2)

Data Center 1



Consumer Group	Offset	Timestamp
my-group-1	10	ts6

Data Center 2



Consumer Group	Offset
my-group-1	6

It is possible for several messages to have the same timestamp. Suppose that a Consumer commits offset 10 in DC1 (i.e. the Consumer has read through message G and has not yet reached H). After offset translation, offset 10 becomes associated with timestamp 6, which corresponds to offsets 5 through 9 in DC2. After failing over to DC2, we know the Consumer has consumed at least one of messages E through I. In this case, the Consumer's offset will be translated to 6, and thus messages F and G will be consumed twice. This guarantees a message won't be skipped, but also opens the possibility that messages will be consumed multiple times during this failover.

# Manual Disaster Recovery: Consumer Group Tool

The command `kafka-consumer-groups` can set a Consumer Group to seek to an offset derived from a timestamp

```
$ kafka-consumer-groups \
  --bootstrap-server dc2-broker-101:9092 \
  --reset-offsets \
  --topic dc1-topic \
  --group my-group \
  --execute \
  --to-datetime 2017-08-01T17:14:23.933
```

tejaswin.renugunta@walgreens.com

# Manual Disaster Recovery: Java Client API

Producer API:

- Replication process must use same partitioner class to preserve message ordering for keyed messages
- Topic properties must be same in source and destination clusters (automatic with Replicator)

Consumer API:

- Use `offsetsForTimes()` method to find an offset for a given timestamp
- Use `seek()` to move to the desired offset

---

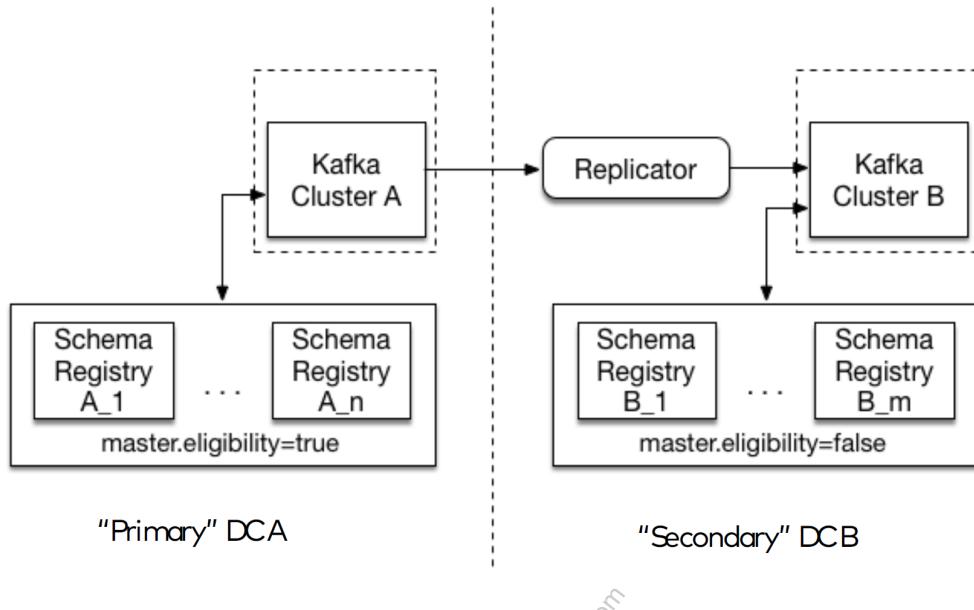
If, for whatever reason, Consumers are not configured with the Replicator interceptor for offset translation, then destination offset will have to be set manually so Consumers at the failover site can recover to an offset relatively close to where they left off before the site failure. The Kafka Client has methods (`offsetsForTimes()`) together with `seek()` which can be embedded in the Consumer code to provide this functionality on a per-Consumer basis. An example of this code is given in the appendix of: <https://www.confluent.io/wp-content/uploads/Disaster-Recovery-Multi-Datacenter-Apache-Kafka-Deployments.pdf>.

Essentially:

1. Maintain a `RESET_TIME` variable that is a safe timestamp to rewind to in case of failure.
2. Create/update a hash map that associates that `RESET_TIME` with each Partition.
3. Connect to the failover Kafka cluster.
4. For each Partition, seek the offset in the new cluster that corresponds to the `RESET_TIME`.

However, in many cases, that would not be a viable option during a recovery if the feature was not already implemented in the client code. As of Kafka 0.11.0, the `kafka-consumer-groups` command has the ability to reset the entire Consumer Group to the offsets corresponding to specific timestamps within the Partitions. See the next slide for an example of this command.

# Schema Registry Across Data Centers



The Schema Registry must be consistent across the datacenters if the schema IDs attached to the messages are to be interpreted properly. This is achieved by extending the primary/secondary relationship of clustered Schema Registry systems to the data center level. Only Primary nodes in the Master data center can update the schema Topic. This ensures a "one voice of truth" model so that the data is consistent across instances.

# Improving Network Utilization

- If network latency is high, increase the TCP socket buffer size in Kafka
  - `socket.send.buffer.bytes` on the origin cluster's Broker (default: 102400 bytes)
  - `receive.buffer.bytes` on Replicator's Consumer (default: 65536 bytes)
  - Increase corresponding OS socket buffer size

---

If the network between the two sites is slow, you may have to tune the network buffers, both in the Broker and OS settings.

The syntax for increasing OS socket buffer size depends on OS type. For example, on CentOS it is in `/proc/sys/net/ipv4/tcp_rmem` and `/proc/sys/net/ipv4/tcp_wmem`.

Enable logging (`log4j.logger.org.apache.kafka.common.network.Selector=DEBUG`) to double check these changes actually took effect. There are instances where the OS silently overrode/ignored settings

tejaswin.renugunta@walgreens.co

# Module Review



- Consider design practices for all components in the Kafka cluster
- ZooKeeper is mission-critical for Kafka clusters
- Kafka can be deployed in multiple data centers

tejaswin.renugunta@walgreens.com

# 09: Conclusion



CONFLUENT

tejaswin.renugunta@walgreens.com

# Course Review

In this course you have learned:



- to describe how the Kafka Brokers, Producers, and Consumers work
- to explain how replication works within the cluster
- to list the essential hardware and runtime configuration options
- to monitor and administer your Kafka Cluster
- to secure your Kafka Cluster
- to integrate Kafka with external systems using Kafka Connect
- to design your Kafka Cluster for high availability and fault tolerance

---

Image from: <http://peter.baumgartner.name/wp-content/uploads/2014/05/peer-review.jpg> (license: creative commons)

# Other Confluent Training Courses

- Confluent Advanced Skills for Optimizing Apache Kafka®
- Confluent Developer Skills for Building Apache Kafka®
- Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB



For more details, see <https://confluent.io/training>

---

- **Confluent Advanced Skills for Optimizing Apache Kafka®**
  - Formulate the Apache Kafka® Confluent Platform specific needs of your organization
  - Monitor all essential aspects of your Confluent Platform
  - Tune the Confluent Platform according to your specific needs
  - Provide first level production support for your Confluent Platform
- **Confluent Developer Skills for Building Apache Kafka®** covers:
  - Fundamentals of Apache Kafka
  - Producing Messages to Kafka
  - Consuming Messages from Kafka
  - Schema Management in Apache Kafka
  - Stream Processing with Kafka Streams
  - Data Pipelines with Kafka Connect
  - Event Streaming Apps with ksqlDB
  - Design Decisions
  - Confluent Cloud
- **Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB** covers:
  - Identify common patterns and use cases for real-time stream processing
  - Understand the high level architecture of Kafka Streams
  - Write real-time applications with the Kafka Streams API to filter, transform, enrich, aggregate, and join data streams

- Describe how ksqlDB combines the elastic, fault-tolerant, high-performance stream processing capabilities of Kafka Streams with the simplicity of a SQL-like syntax
- Author ksqlDB queries that showcase its balance of power and simplicity
- Test, secure, deploy, and monitor Kafka Streams applications and ksqlDB queries

tejaswin.renugunta@walgreens.com

# Confluent Certified Administrator for Apache Kafka

**Duration:** 90 minutes

**Qualifications:** Solid work foundation in Confluent products and 6-to-9 months hands-on experience

**Availability:** Live, online, 24-hours per day!

**Cost:** \$150

**Register online:** [www.confluent.io/certification](http://www.confluent.io/certification)



---

This course prepares you to manage a production-level Kafka environment, but does not guarantee success on the Confluent Certified Administrator Certification exam. We recommend running Kafka in Production for a few months and studying these materials thoroughly before attempting the exam.

## Benefits:

- Recognition for your Confluent skills with an official credential
- Digital certificate and use of the official Confluent Certified Administrator Associate logo

## Exam Details:

- The exam is linked to the current Confluent Platform version
- Multiple choice and multiple select questions
- 90 minutes
- Designed to validate professionals with a minimum of 6 - 12 months of Confluent experience
- Remotely proctored on your computer
- Available globally in English

# Confluent Certified Developer for Apache Kafka

**Duration:** 90 minutes

**Qualifications:** Solid work foundation in Confluent products and 6-to-12 months hands-on experience

**Availability:** Live, online, 24-hours a day!

**Cost:** \$150

**Register online:** [www.confluent.io/certification](http://www.confluent.io/certification)



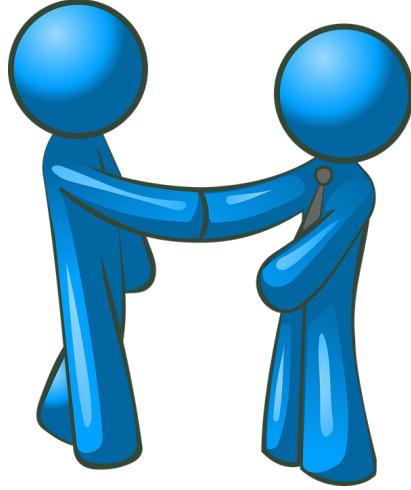
## Benefits:

- Recognition for your Confluent skills with an official credential
- Digital certificate and use of the official Confluent Certified Developer Associate logo

### Exam Details:

- The exam is linked to the current Confluent Platform version
- 60 multiple choice questions in 90 minutes
- Designed to validate professionals with a minimum of 6 - 12 months of Confluent experience
- Remotely proctored on your computer
- Available globally in English

Thank You!

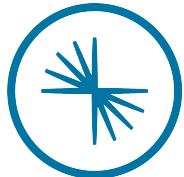


- Thank you for attending the course!
- Please complete the course survey
- For any feedback email [training-admin@confluent.io](mailto:training-admin@confluent.io)

- 
- Survey: your instructor will give you details on how to access the survey
  - Feedback: Any feedback is welcome. Please do not hesitate to contact us at the given email address.

tejaswin.renugunta@walgreens.com

# Appendix A: Important JMX Metrics



CONFLUENT

tejaswin.renugunta@walgreens.com

# About These Metrics

- Kafka provides many metrics for monitoring
- The metrics provided here are a subset of the metrics available
- Consult the Kafka documentation for a complete list

tejaswin.renugunta@walgreens.com

# Kafka Up and Running

- BrokerState
  - kafka.server:type=KafkaServer, name=BrokerState
  - Kafka broker state
  - In steady state, this should be "RunningAsBroker"
- ZooKeeperExpiresPerSec
  - kafka.server:type=SessionExpireListener, name=ZooKeeperExpiresPerSec
  - Normal value is 0

tejaswin.renugunta@walgreens.com

# Controller and Partitions

- `ActiveControllerCount`
  - `kafka.controller:type=KafkaController, name=ActiveControllerCount`
  - Number of active Controllers
  - Only one Broker should have the value `1`, the rest should have value `0`
- `OfflinePartitionsCount`
  - `kafka.controller:type=KafkaController, name=OfflinePartitionsCount`
  - Number of Partitions that do not have an active leader and hence are not writable or readable
  - Alert if the value is greater than `0`

tejaswin.renugunta@walgreens.com

# Replicas and Partitions

- `UnderReplicatedPartitions`
  - `kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions`
  - Number of under-replicated Partitions ( $| ISR | < | all replicas |$ )
  - Alert if the value is greater than 0
- `PartitionCount`
  - `kafka.server:type=ReplicaManager, name=PartitionCount`
  - Number of Partitions on this Broker
  - This should be relatively even across all Brokers

tejaswin.renugunta@walgreens.com

# Replica Leaders

- LeaderCount
  - `kafka.server:type=ReplicaManager, name=LeaderCount`
  - Number of leaders on this Broker
  - This should be approximately even across all Brokers. If not, set `auto.leader.rebalance.enable` to `true` on all Brokers in the cluster

tejaswin.renugunta@walgreens.com

# In-Sync Replicas

- **IsrExpandsPerSec**
  - `kafka.server:type=ReplicaManager, name=IsrExpandsPerSec`
  - When a Broker is brought up after a failure, it starts catching up by reading from the Leader. Once it is caught up, it gets added back to the ISR
- **IsrShrinksPerSec**
  - `kafka.server:type=ReplicaManager, name=IsrShrinksPerSec`
  - If a Broker goes down, ISR for some of the Partitions will shrink. When that Broker is up again, ISR will be expanded once the replicas are fully caught up
  - Other than that, the expected value for both ISR shrink rate and expansion rate is 0. Too frequent often indicates a garbage collection issue

tejaswin.renugunta@walgreens.com

# Leader Elections

- UncleanLeaderElectionsPerSec
  - kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec
  - Unclean leader election rate (Meter)
- LeaderElectionRateAndTimeMs
  - kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs
  - Leader election rate (Meter)
- MaxLag
  - kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica
  - Maximum lag between Followers and Leaders, Useful for MirrorMaker (Gauge)

tejaswin.renugunta@walgreens.com

# Topic-Level Attributes

- `kafka.producer:type=producer-topic-metrics,client-id=producer-1,topic=topic_name`
  - `byte-rate`
  - `record-send-rate`
  - `record-error-rate`
  - `compression-rate-avg`

tejaswin.renugunta@walgreens.com

# Producer-Level Attributes

- `kafka.producer:type=producer-metrics,client-id=client_id`
  - `batch-size-avg`
  - `compression-rate-avg`
  - `io-ratio`
  - `io-wait-ratio`

tejaswin.renugunta@walgreens.com

# Consumer-Level Attributes

- `kafka.consumer:type=consumer-fetch-manager-metric,client-id=client_id`
  - `records-lag-max`
    - Should be 0
  - `fetch-rate`
    - Should be > 0
  - `records-consumed-rate`
  - `bytes-consumed-rate`

tejaswin.renugunta@walgreens.com

# Log Compaction

- max-dirty-percent
  - kafka.log:type=LogCleanerManager, name=max-dirty-percent
  - Max dirty percent
- cleaner-recopy-percent
  - kafka.log:type=LogCleaner, name=cleaner-recopy-percent
  - Cleaner recopy percent
- max-clean-time-secs
  - kafka.log:type=LogCleaner, name=max-clean-time-secs
  - Last cleaning time

tejaswin.renugunta@walgreens.com

# Throughput Per Topic

- BytesInPerSec
  - kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
- ReplicationBytesInPerSec
  - kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesInPerSec
- MessagesInPerSec
  - kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
- BytesOutPerSec
  - kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
- ReplicationBytesOutPerSec
  - kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesOutPerSec

tejaswin.renugunta@walgreens.com

# Thread Capacity

- RequestHandlerAvgIdlePercent
  - kafka.server:type=KafkaRequestHandlerPool, name=RequestHandlerAvgIdlePercent
  - I/O threads and network threads
- NetworkProcessorAvgIdlePercent
  - kafka.network:type=SocketServer, name=NetworkProcessorAvgIdlePercent
  - Values are between 0 and 1

tejaswin.renugunta@walgreens.com

# Latency

- Produce/Fetch request time (Histograms)
  - `kafka.network:type=RequestMetrics, name=TotalTimeMs, request=Produce`
  - `kafka.network:type=RequestMetrics, name=TotalTimeMs, request=FetchConsumer`
  - `kafka.network:type=RequestMetrics, name=TotalTimeMs, request=FetchFollowe`
- Breakdown (Histograms)
  - `kafka.network:type=RequestMetrics, name=LocalTimeMs, request=Produce`
  - `kafka.network:type=RequestMetrics, name=RemoteTimeMs, request=Produce`
  - `kafka.network:type=RequestMetrics, name=ResponseQueueTimeMs, request=Prod`
  - `uce`
- Request Queue Size
  - `kafka.network:type=RequestChannel, name=RequestQueueSize`

tejaswin.renugunta@walgreens.com

# Quota

- Broker side:

```
kafka.server:type={Produce|Fetch},client-id=([-.\w]+)
```

- Attribute `throttle-time` indicates the amount of time in milliseconds the client-id was throttled (0 if not throttled)
- Attribute `byte-rate` indicates the data produce/consume rate of the client in bytes/second

- Producer side:

```
kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+)
```

- Attributes `produce-throttle-time-max` and `produce-throttle-time-avg`: Maximum and average times in milliseconds a request has been throttled

- Consumer side:

```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.\w]+)
```

- Attributes `fetch-throttle-time-max` and `fetch-throttle-time-avg`: Maximum and average times in milliseconds a request has been throttled

tejaswin.renugunta@walgreens.com

# MirrorMaker

- `bufferpool-wait-ratio`
  - `kafka.producer:type=producer-metrics,name={client-id} bufferpool-wait-ratio`
  - The time an appender waits for space allocation. Should be low
- `ResponseSendTimeMs`
  - `kafka.network:type=RequestMetrics, name=ResponseSendTimeMs`
  - Response send time. A high value can indicate slow zero-copy performance or a slow network (Histogram)

tejaswin.renugunta@walgreens.com

# Module Review

- Collect important metrics to monitor the health of your Kafka cluster
- Consult the Kafka documentation for a complete list

tejaswin.renugunta@walgreens.com