

# Apache Kafka® Administration

## By Confluent

Version 7.5.2-v1.0.0



CONFLUENT

*hitesh@dataconfluent.io*

# Table of Contents

<b>Introduction</b> . . . . .	<b>1</b>
Class Logistics and Overview	2
Fundamentals Review	9
<b>1: Bridging From Fundamentals</b> . . . . .	<b>12</b>
1a: How Can You Leverage Replication? . . . . .	16
Lab: Introduction . . . . .	25
Lab: Using Kafka's Command Line Tools . . . . .	26
Lab: Producing Records with a Null Key . . . . .	27
<b>2: Producing Messages Reliably</b> . . . . .	<b>28</b>
2a: How Do Producers Know Brokers Received Messages? . . . . .	31
2b: How Can Kafka recognize Duplicates caused by Retries? . . . . .	38
2c: How Does Kafka Handle the Notion of Producers Sending Messages in Transactions? . . . . .	41
<b>3: Replicating Data: A Deeper Dive</b> . . . . .	<b>47</b>
3a: How Does Kafka Determine Which Messages Can be Consumed? . . . . .	50
3b: How Does Kafka Place Replicas and How Can You Control Replication Further? . . . . .	59
3c: How Does Kafka React When a Leader Dies? . . . . .	68
3d: How Does Kafka Track Leadership Changes? . . . . .	72
3e: How Does Kafka Track Follower Responsiveness? . . . . .	82
<b>4: Providing Durability in Other Ways</b> . . . . .	<b>85</b>
4a: How Does Kafka Organize Files to Store Partition Data? . . . . .	88
Lab: Investigating the Distributed Log . . . . .	95
4b: What are the Basics of Scaling Consumption? . . . . .	96
4c: How Does Kafka Maintain Consumer Offsets? . . . . .	101
<b>5: Configuring a Kafka Cluster</b> . . . . .	<b>107</b>
5a: How Do You Configure Brokers? . . . . .	110
5b: What if You Want to Adjust Settings Dynamically or Apply at the Topic Level? . . . . .	118
Lab: Exploring Configuration . . . . .	130
Lab: Increasing Replication Factor . . . . .	131
<b>6: Managing a Kafka Cluster</b> . . . . .	<b>132</b>
6a: What Should You Consider When Installing and Upgrading Kafka? . . . . .	135
6b: What is a Controller vs a Broker? . . . . .	142
6c: What are the Basics of Monitoring Kafka? . . . . .	148
6d: How Can You Decide How Kafka Keeps Messages? . . . . .	157
6e: How Can You Move Partitions To New Brokers Easily? . . . . .	171
6f: What Should You Consider When Shrinking a Cluster? . . . . .	179

Lab: Kafka Administrative Tools .....	182
<b>7: Balancing Load with Consumer Groups and Partitions .....</b>	<b>183</b>
7a: How Do Partitions and Consumers Scale? .....	186
7b: How Do Groups Distribute Work Across Partitions? .....	193
7c: How Does Kafka Manage Groups? .....	201
Lab: Modifying Partitions and Viewing Offsets .....	209
<b>8: Optimizing Kafka's Performance .....</b>	<b>210</b>
8a: How Does Kafka Handle the Idea of Sending Many Messages at Once? .....	214
Lab: Exploring Producer Performance .....	222
8b: How Do Produce and Fetch Requests Get Processed on a Broker? .....	223
8c: How Can You Measure and Control How Requests Make It Through a Broker? .....	231
8d: What Else Can Affect Broker Performance? .....	242
8e: How Do You Control It So One Client Does Not Dominate the Broker Resources? .....	250
8f: What Should You Consider in Assessing Client Performance? .....	259
8g: How Can You Test How Clients Perform? .....	265
Lab: Performance Tuning .....	269
<b>9: Securing a Kafka Cluster .....</b>	<b>270</b>
9a: What are the Basic Ideas You Should Know about Kafka Security? .....	273
9b: What Options Do You Have For Securing a Kafka/Confluent Deployment? .....	282
9c: How Can You Easily Control Who Can Access What? .....	286
9d: What Should You Know Securing a Deployment Beyond Kafka Itself? .....	300
Lab: Securing the Kafka Cluster .....	305
<b>10: Understanding Kafka Connect .....</b>	<b>306</b>
10a: What Can You Do with Kafka Connect? .....	309
10b: How Do You Configure Workers and Connectors? .....	320
10c: Deep Dive into a Connector & Finding Connectors .....	330
10d: What Else Can One Do With Connect? .....	339
Lab: Running Kafka Connect .....	343
<b>11: Deploying Kafka in Production .....</b>	<b>344</b>
11a: What Does Confluent Advise for Deploying Brokers in Production? .....	349
11b: What Does Confluent Advise for Deploying Kafka Connect in Production? .....	367
11c: What Does Confluent Advise for Deploying Schema Registry in Production? .....	371
11d: What Does Confluent Advise for Deploying the REST Proxy in Production? .....	376
11e: What Does Confluent Advise for Deploying Kafka Streams and ksqlDB in Production? .....	380
11f: What Does Confluent Advise for Deploying Control Center in Production? .....	384
<b>Conclusion .....</b>	<b>387</b>
<b>Appendix: Confluent Technical Fundamentals of Apache Kafka® Content .....</b>	<b>395</b>
1: Getting Started .....	397

<b>2: How are Messages Organized? . . . . .</b>	<b>406</b>
<b>3: How Do I Scale and Do More Things With My Data? . . . . .</b>	<b>411</b>
<b>4: What's Going On Inside Kafka? . . . . .</b>	<b>420</b>
<b>5: Recapping and Going Further . . . . .</b>	<b>430</b>
<b>Appendix: Additional Content . . . . .</b>	<b>439</b>
<b>Appendix A: Detailed Transactions Demo . . . . .</b>	<b>441</b>
<b>Appendix B: How Can You Monitor Replication? . . . . .</b>	<b>456</b>
<b>Appendix C: Multi-Region Clusters . . . . .</b>	<b>461</b>
<b>Appendix D: SSL and SASL Details . . . . .</b>	<b>477</b>

hitesh@datacouch.io

# Introduction



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Class Logistics and Overview

## Copyright & Trademarks

Copyright © Confluent, Inc. 2014-2024. [Privacy Policy](#) | [Terms & Conditions](#).

Apache, Apache Kafka, Kafka, and the Kafka logo are trademarks of the  
[Apache Software Foundation](#)

All other trademarks, product names, and company names or logos cited herein  
are the property of their respective owners.

hitesh@datacouch.io

# Prerequisite



This course requires a working knowledge of the Apache 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 <https://training.confluent.io/learningpath/apache-kafka-fundamentals> for anyone who needs to catch up.

# Agenda



This course consists of these modules:

- Bridging From Fundamentals
- Producing Messages Reliably
- Replicating Data: A Deeper Dive
- Providing Durability in Other Ways
- Configuring a Kafka Cluster
- Managing a Kafka Cluster
- Balancing Load with Consumer Groups and Partitions
- Optimizing Kafka's Performance
- Securing a Kafka Cluster
- Understanding Kafka Connect
- Deploying Kafka in Production

hitesh@datacouch.io

# Course Objectives

Upon completion of this course, you should be able to:

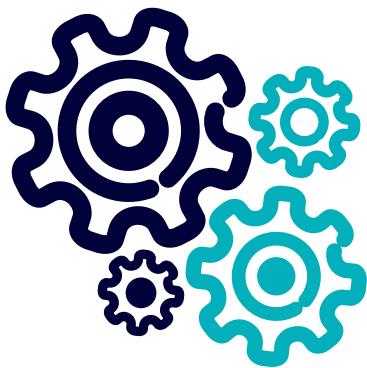
- Describe how Kafka brokers, producers, and consumers work
- Describe how replication works within the cluster
- Understand hardware and runtime configuration options
- Monitor and administer your Kafka cluster
- Integrate Kafka with external systems using Kafka Connect
- Design a Kafka cluster for high availability & fault tolerance

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

---

- "Hands-On Exercises" refers to lab exercises that are designed to follow up on many of the lessons. Your instructor will introduce you to our lab environment. These exercises are in the Exercise Guide, and there are slides in your handbook at the earliest point you are enabled to do them.
- "Activities" refers to assorted interactive aspects of the course. Such activities appear at the end of some lessons and embedded in others. Your instructor may choose to approach these in various ways, but they are included in the content to give you opportunities to engage with and reinforce the material.

# Class Logistics



- Timing
  - Start and end times
  - Can I come in early/stay late?
  - Breaks
  - Lunch
- Physical Class Concerns
  - Restrooms
  - Wi-Fi and other information
  - Emergency procedures
  - Don't leave belongings unattended



No recording, please!

---

Expanding on the rule at the bottom: You are not permitted to record via any medium, or stream via any medium any of the content from this class.

## How to get the courseware?

1. Register at **training.confluent.io**
2. Verify your email
3. Log in to **training.confluent.io** and enter your **license activation key**
4. Go to the **Classes** dashboard and select your class



---

Your instructor may choose to have you do this now, combine it with the first lab, or do it before class begins.

# Introductions



## About you:

- What is your name, your company, and your role?
- Where are you located (city, timezone)?
- What is your experience with Kafka?
- Which other Confluent courses have you attended, if any?
- (For Administration classes) What is your experience with Linux? With container technology?
- (For Developer classes) Which Computer Languages are you fluent with?

## About your instructor

hitesh@datacouch.io

# Fundamentals Review

## Discussion

### Question Set 1 [4 minutes]

Determine if each statement is true or false and why:

1. All messages in a topic are on the same broker.
2. All messages in a partition are on the same broker.
3. In a topic, all messages that have the same key will be on the same broker.
4. The more partitions a topic has, the higher the performance that can be achieved.

### Question Set 2 [4 minutes]

Determine the best answer to each question.

1. What are the roles of a producer and a consumer?
2. How is it decided which messages consumers read?
3. Who initiates the reading of messages: consumers or the Kafka cluster?
4. How many times can a single message be consumed?



How your instructor approaches this section may vary depending on the particular class.

### Answers to Question Set 1

1. Mostly false. Kafka tries to balance the workload by selecting different brokers to "host" different partitions (we say that a broker is the leader of that partition). Because messages of the same topic are sent to different partitions, they can land on different topics. The exceptions are when the topic has a single partition and when the cluster is badly balanced (one broker is the leader of all partitions of the topic).
2. True by definition. For each partition there is a broker that acts as its leader: if a message is not in the leader then it is not in the partition. When we add replication, there can be other brokers, (which we call followers) that may be lagging behind the leader, so they don't have all the messages (yet).
3. True by default. For keyed messages, partition placement is determined by `hash(key) % num_partitions` by default, so all messages of a given key will always land on the same partition (as long as the number of partitions does not change), and thus on the same broker.

- True up to certain point. A topic with more partitions allows for more parallelism at the broker- and consumer-level, there are diminishing returns because of increased overhead.

## Answers to Question Set 2

- A producer has to serialize the payload of the message, to select the target partition, and to send the message to the Kafka cluster. A consumer has to subscribe to topics, keep track of its offsets, fetch messages from the Kafka cluster based on said offsets, and potentially work alongside other consumers in a cooperative way by taking advantage of the parallelism that partitions allow.
- Consumers must keep track of the offset of the next message they want to read per partition (what we call consumer offsets), then typically read messages sequentially.
- Kafka is a **poll** system, so consumers initiate the reading.
- Consumers works independently, unless they are cooperating with each other in what we call consumer groups. So if one consumer in a consumer group reads a message, no other consumer in the same group can read it (this is done for parallelism). On the other side, consumers belonging to different groups may read the same message, even at different times! Expert mode: how the code persists offsets—offset commit—may change this, or if the code manipulates the offsets.

# Instructor-Led Review

Some time is allocated here for an instructor-led review/Q&A on prerequisite concepts from Fundamentals.

hitesh@datacouch.io

# 1: Bridging From Fundamentals



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 1 lesson:

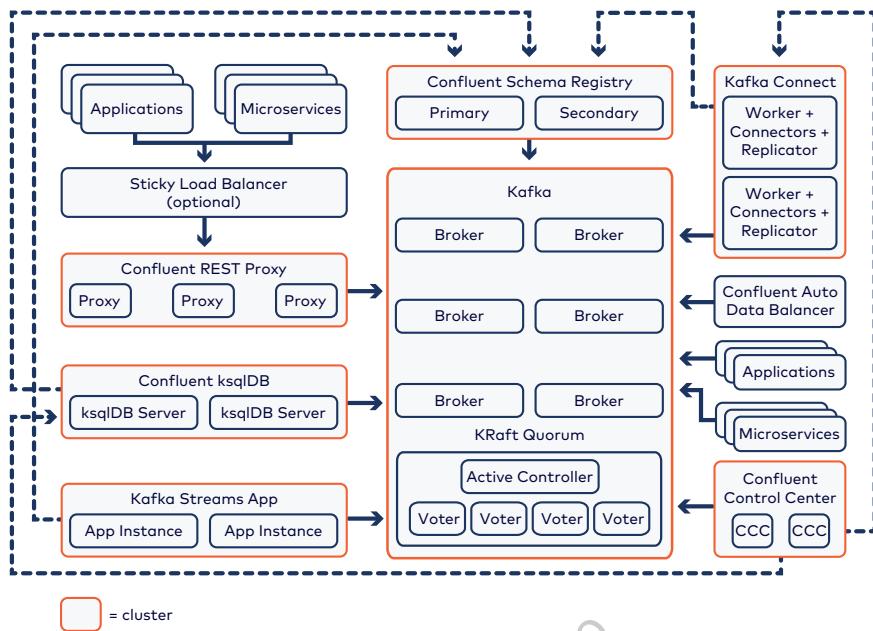
- How Can You Leverage Replication?

Where this fits in:

- Recommended Prerequisite: Fundamentals course

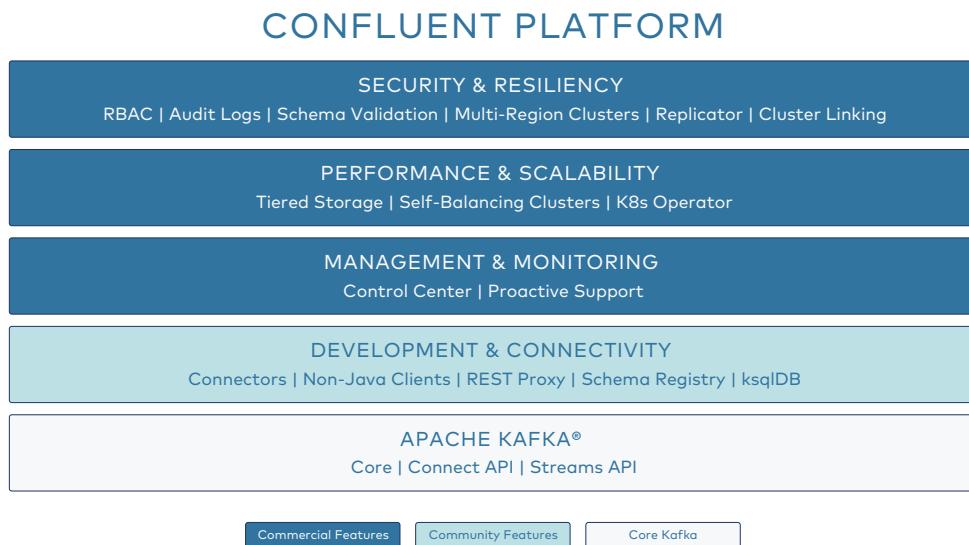
hitesh@datacouch.io

# Kafka Deployment Architecture



This is a diagram of a possible Kafka/Confluent deployment. It is given to you now to show you the "big picture."

# What Does Confluent Platform Add to Kafka?



Confluent Platform adds additional features beyond the core. The top two boxes in the medium shade of blue are paid features; the next two in the teal shade of blue are free features.

You can also view [a more-involved version of the graphic from our current documentation](#).

# 1a: How Can You Leverage Replication?

## Description

Review of leaders vs. followers. Replication factor. How messages get from leaders to followers and config. ISRs. Leader failover / leader election.

hitesh@datacouch.io

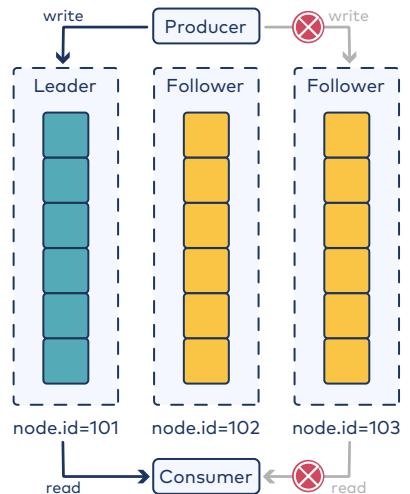
# Review: Basics of Replication

Ensure high availability of data with multiple **replicas** of partitions

**Leader** Always one per partition.  
Clients connect to it to write and read

**Follower** Generally multiple per partition.  
They keep extra copies from the leader

Topic setting `replication.factor`

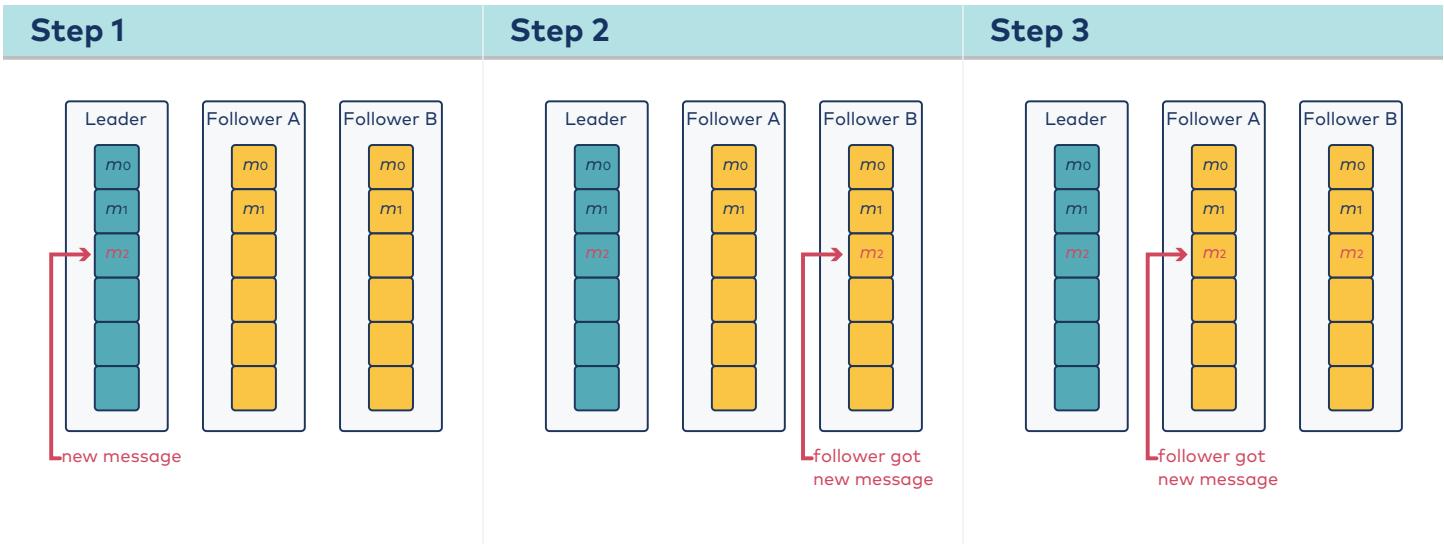


In the picture, `replication.factor = 3`.

Observe that producers write **only** to the leader, never followers.

Observe that consumers read **only** from the leader, not from followers (later we may introduce Follower Fetching).

# "Follow the Leader"

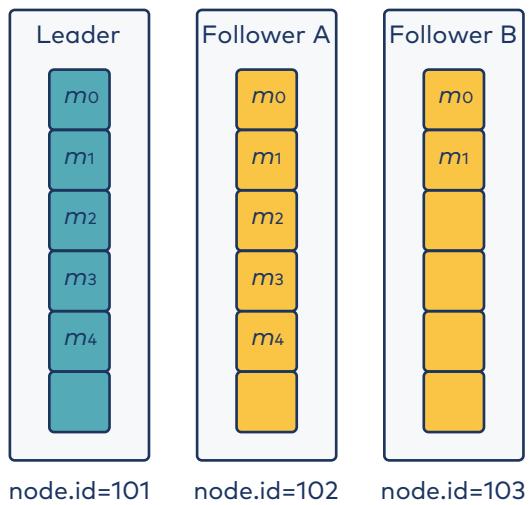


Observe the three steps that are illustrated:

1. We start with a leader that has two messages. The two followers shown are caught up, i.e. they have the same two messages. A new message is written to this partition, and it goes to the leader.
2. The followers are monitoring the leader, periodically checking for new messages. One follower gets the new message.
3. Then a second follower gets the new message.

(Both followers may be perfectly timed with each other and Steps 2 and 3 happen simultaneously, but they do not have to.)

## But Observe:



- Follower A (on node 102) is an **in-sync replica (ISR)**
- Follower B (on node 103) is **not**

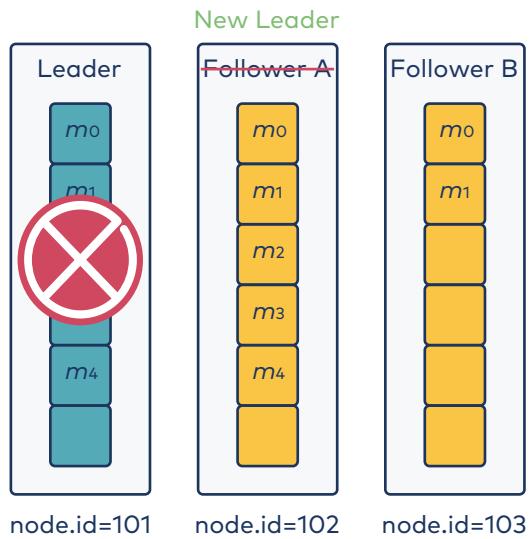
---

Note that the leader is always in-sync with itself and is always considered an in-sync replica.

Followers that are not ISRs are often referred to as stuck followers.

Note that replicas **are** partitions. It wouldn't make sense to have more than one replica on the same node, so tools reporting replica information can use node IDs to reference replicas. Given that, we would often see the ISR list for this picture written as [101, 102].

# Leader Failover



**Question:** Would either choice of follower have been equally good to replace the leader that had died?

---

Answer: No, an in-sync replica is the best choice of a new leader. We want the (new) leader to be caught up with the old leader. It's stepping in to do the job the old leader was doing.

# How Does Kafka Choose Leaders?

- Leader election happens automatically
  - Kafka will generally choose an in-sync follower to become leader
  - Leader election does not happen in parallel
  - Background processes manage balance of leadership
- 

It's important for you to know that when a broker containing a leader goes down, a follower will become the new leader. But, you don't need to worry about how this happens; Kafka will take care of this for you.

One thing that's important to know, however, is that the leader election process cannot be parallelized. So, if a broker that contains the leaders for four different partitions goes down, Kafka will need to choose a new leader for the first, then the second, then the third, and then the fourth—in succession and not in parallel. Having leaders spread evenly across brokers is thus good. Administrators should monitor for this, but Kafka has processes in place to help with it.

In case you're curious:

- For each partition, there is a preferred replica, or a broker where having its leader would yield the best balance. Kafka has background processes that monitor how many leaders are not in the preferred place, and, when a threshold has been crossed, Kafka balances this.
- One Kafka node acts as Active Controller, and it handles leader election.
- If there is no in-sync follower able to become leader, Kafka will not select an out-of-sync follower, as this could lead to data loss. Administrators may choose to turn this on, though, understanding the implications, and allowing for something called "unclean leader election."

# Configuring Replication Factor

Increase the replication factor for better durability guarantees

- When creating a topic:

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

- Cluster-wide default value in `server.properties` (on all Kafka nodes)
  - `default.replication.factor` (Default: 1)

---

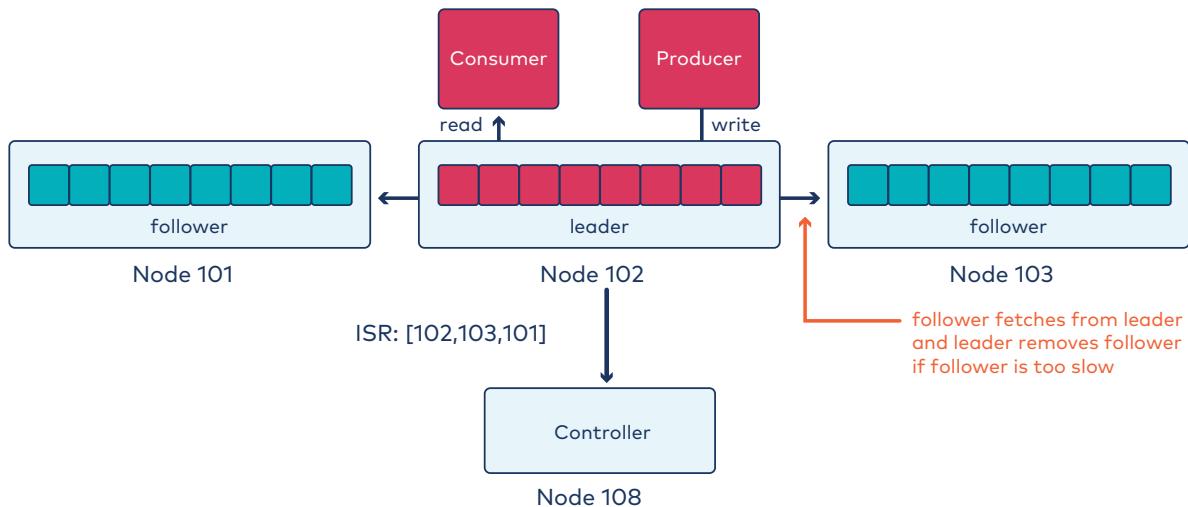
There are 2 basic ways to create a topic:

- \* Through the Admin API library. This library can be called by your applications, and it is also used by Confluent Control Center and the `kafka-topics` CLI. In this case we can specify the replication factor, or it will be created with the default cluster-wide value.
- \* So-called *automatic* topic creation. This functionality is enabled by default: when a producer application sends a Produce request to Kafka for a topic that doesn't exist, Kafka will automatically create that topic with default values (number of partitions, replication factor, clean-up policy, etc.)

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).

Also, notice that we do not specify which brokers to use; in most cases it is best to let Kafka decide where to place replicas.

# 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 the cluster metadata (through the Controller).
- If the leader fails, it is the list of ISRs which is used by the Controller to elect a new leader.

# Activity: Exploring Replica Placement & Replication Behavior



Say we have 5 brokers -  $b_1, b_2, \dots, b_5$ .

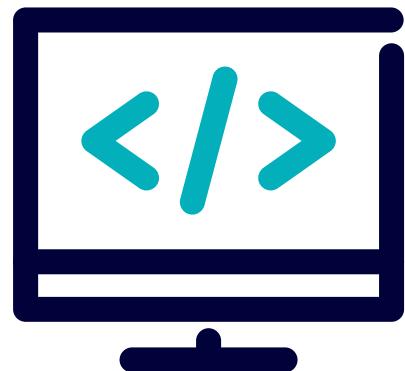
Say we have a replication factor of 4.

1. How many followers would we have?
2. Say leader is on broker  $b_5$ .
  - a. Where could the followers be?
  - b. Where could a follower **not** be?
3. Say we have 3 successfully written messages that have been properly replicated. It's time to write the fourth message.
  - a. Where does it go?
  - b. What happens next?
4. Say broker  $b_5$  fails. What happens? Why?

# Lab: Introduction

Please work on **Lab 1a: Introduction**

Refer to the Exercise Guide

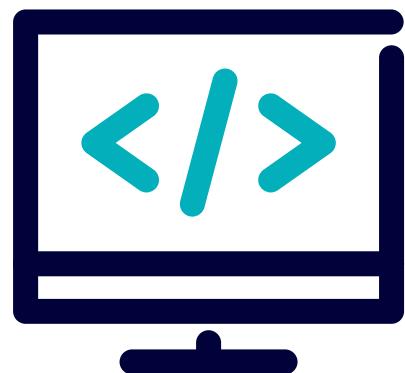


hitesh@datacouch.io

# Lab: Using Kafka's Command Line Tools

Please work on **Lab 1b: Using Kafka's Command Line Tools**

Refer to the Exercise Guide

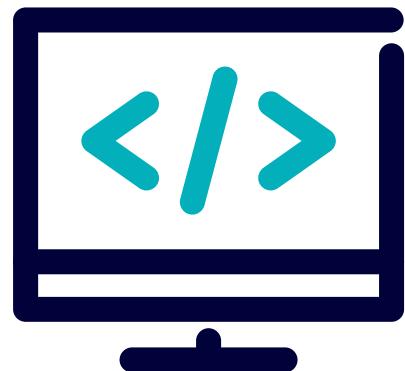


hitesh@datacouch.io

# Lab: Producing Records with a Null Key

Please work on **Lab 1c: Producing Records with a Null Key**

Refer to the Exercise Guide



hitesh@datacouch.io

# 2: Producing Messages Reliably



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 3 lessons:

- How Do Producers Know Brokers Received Messages?
- How Can Kafka recognize Duplicates caused by Retries?
- How are Transactional Messages Tagged and Handled?

Where this fits in:

- Hard Prerequisite: Fundamentals course

hitesh@datacouch.io

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Explain how producers reliably send messages to the brokers
- Explain how Kafka achieves Exactly Once Semantics (EOS)

hitesh@datacouch.io

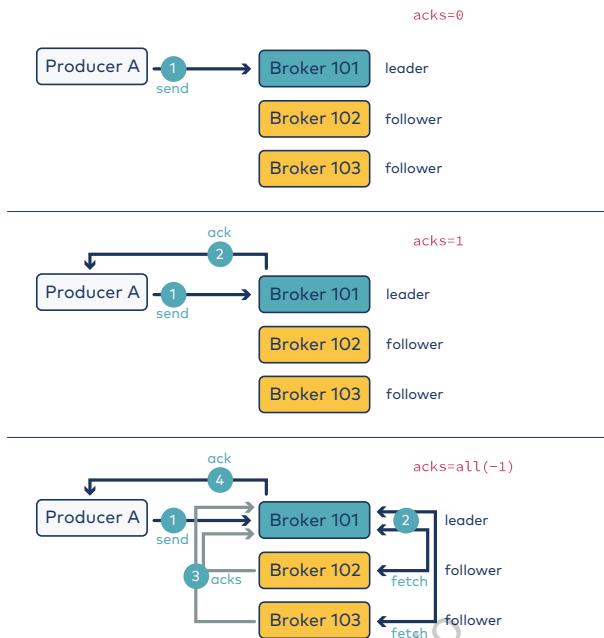
## 2a: How Do Producers Know Brokers Received Messages?

### Description

Producer acknowledgements.

hitesh@datacouch.io

# acks - Three Cases, Ideal Performance



We have three choices for the `acks` setting:

- `0` means the Kafka cluster does **not** communicate back to the producer whether a message has been received.
- `1` means that once the leader has persisted the message, it communicates an acknowledgement back to the producer.
- `all` means that once **leader and all** followers have persisted the message, the leader communicates an acknowledgement back to the producer.

Put differently, `1` means that an ack is sent after record is stored in "1" member of the ISR, whereas `all` means that an ack is sent after the record is stored in "all" members of the ISR.

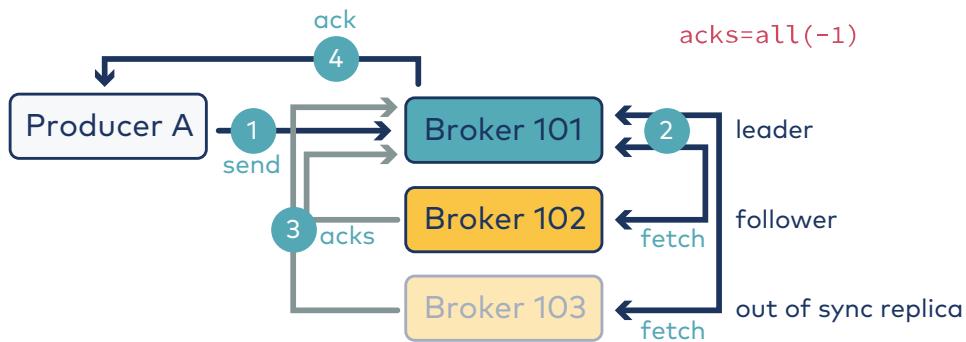
This loosely relates to message delivery guarantees. Note that there are three cases:

- At most once: Messages may be lost but are never redelivered.
- At least once: Messages are never lost but may be redelivered.
- Exactly once: this is what people generally want; each message is delivered once and only once.

There is a later full lesson on delivery guarantees in the Advanced Concepts branch of the course.

hitesh@datacouch.io

...But not all followers are in-sync...



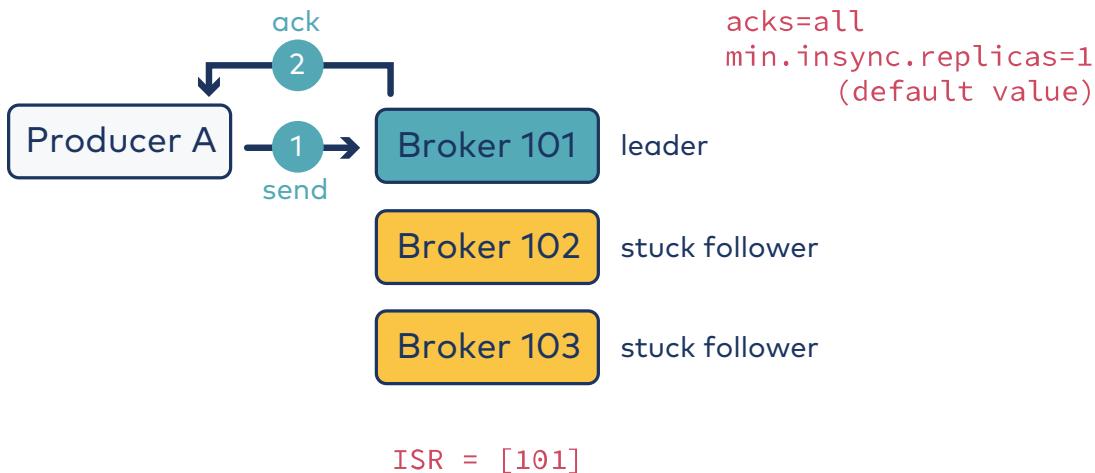
So... when the leader gets a new message and `acks=all`...

1. It notes which followers are **in sync** with the leader when it receives the message
2. Followers fetch from the leader and send acks to the leader
3. When the leader gets acks from all followers from (1), it sends an ack to the producer

---

If we took the `acks = all` requirement from the last slide literally, Kafka would require stuck followers not only to get the new message, but also catch up on prior messages in order to satisfy the `acks` request. Instead, Kafka does **not** require stuck followers to catch up to satisfy `acks = all`; the new message must only be received by followers that were in-sync with the leader at the time the leader received the new message.

# What if We Don't Have Any In-Sync Followers?

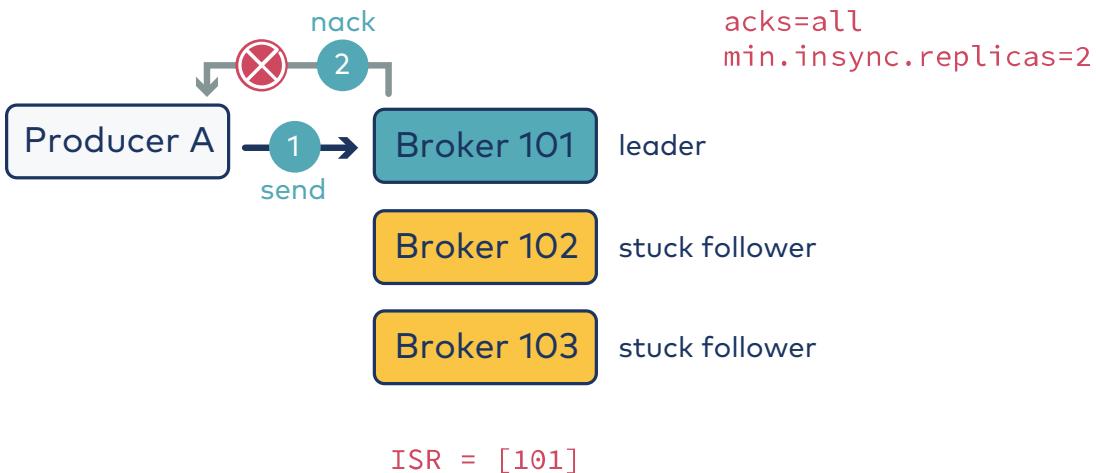


Let's be literal about what `acks = all` requires:

1. The leader must persist the new message
2. All followers that were in sync with the leader at the time the leader received the message must also receive the message.

What if no followers were in sync with the leader? Then the second condition is vacuously true and `acks = all` is met. But if one has set `acks = all`, that means the producer wants certainty that the message has been delivered to one or more followers. This isn't so good...

# Guaranteeing Meaningful `acks=all`

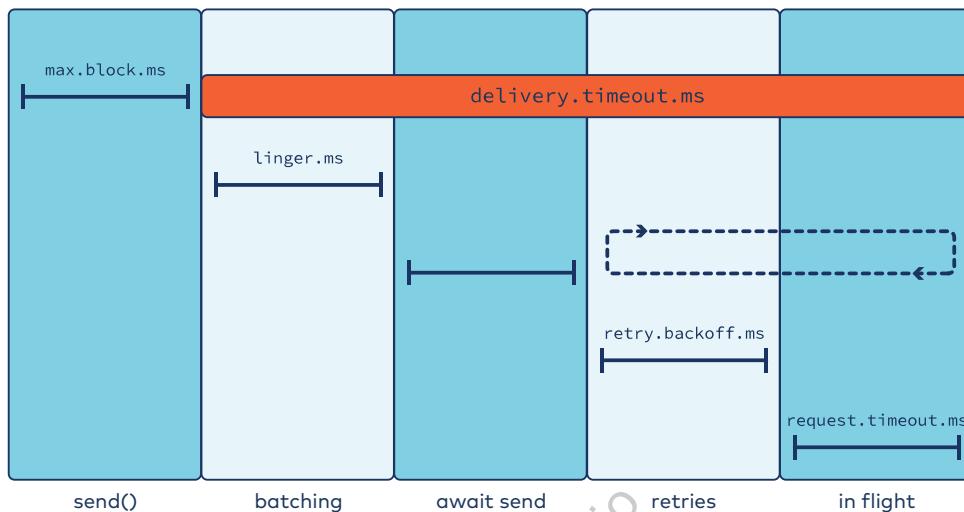


We'd often like `acks = all` to be stronger and only be met when a new message has made it to at least one follower. We can strengthen it by setting `min.insync.replicas` to `2` or greater. (Remember, the leader always counts as one in-sync replica, so this value must be strictly greater than 1.)

Also, note that while `acks` is a producer setting, `min.insync.replicas` is a broker setting.



## Producer Time Limits



Name	Description	Default
<code>retries</code>	Number of times producer will retry sending messages	<code>MAX_INT</code>
<code>request.timeout.ms</code>	An upper bound on the time a producer will wait to hear acknowledgments back from the cluster.	30 sec.
<code>retry.backoff.ms</code>	How much time is added after a failed request before retrying it.	100
<code>delivery.timeout.ms</code>	An upper bound on the time to report success or failure after a call to <code>send()</code> returns. Use this to control producer retries.	2 minute s.



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

You can review a larger list of producer configurations on our website.

## 2b: How Can Kafka recognize Duplicates caused by Retries?

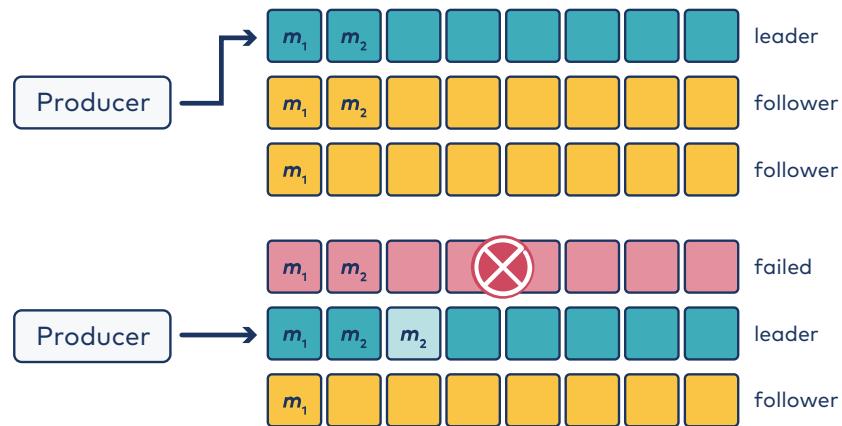
### Description

Idempotent producers: why, metadata, benefits.

hitesh@datacouch.io

# Problem: Producing Duplicates to the Log

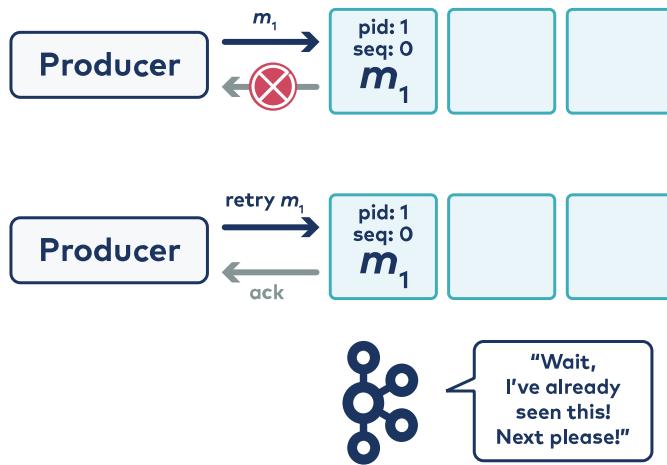
- `acks = all`
- `retries > 0`



Here, the producer has settings `acks=all` with `retries` enabled. The leader fails before record batch  $m_2$  is replicated by all the brokers. The new leader already received  $m_2$  from the previous leader, not the producer, so it doesn't know it has to acknowledge. Because the new leader won't send the acknowledgement, the producer's retry loop will send a new request, causing  $m_2$  to be duplicated.

# Solution: Idempotent Producers

- `enable.idempotence = true`
- `acks = all`



The `enable.idempotence = true` setting in the Producer ensures messages aren't duplicated, even in the case of Producer retries or Broker failure. This is possible due to headers in the message format for producer ID and sequence number.

- **Producer ID:** A unique identifier for a producer session
- **Sequence number:** Each message a producer sends is given a sequence number that increments with each message.

In this example, we see that the broker recognizes the sequence number from this producer, so it acknowledges the producer without appending a duplicate of  $m_1$  to the log.

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  $\text{PID} \rightarrow \text{Sequence number}$ , but this could take a while. The `.snapshot` file speeds up this process.



Enabling idempotent producers usually has negligible performance impact, thus making it useful in many situations. 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.

# 2c: How Does Kafka Handle the Notion of Producers Sending Messages in Transactions?

## Description

Transactions. How to enable on producers. Metadata. Effects. How brokers handle messages in transactions. How consumers handle transactional messages.

hitesh@datacouch.io

# 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 transactions)
- Sample use cases:
  - tracking ad views for billing
  - processing financial transactions
  - tracking inventory in the supply chain

---

More info: [How Kafka Achieves EOS](#)

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

# 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

# Enabling Exactly Once Semantics

Two components:

## 1. Idempotent Producers

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

## 2. Transactions

- Producer:
  - Set a unique `transactional.id` for each producer instance
  - Use the transactions API in the producer code

- Consumer:
  - Set `isolation.level = read_committed` on the consumer

The **Kafka Streams API** was designed with exactly-once processing in mind:  
`processing.guarantee=exactly_once`

---

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.

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.

The configuration in a Kafka Streams is simpler, based on a single parameter `processing.guarantee=exactly_once` to get exactly once processing (Default: `at_least_once`.)

# 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

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 `read_committed` Consumers must wait for the amount of time specified by the Producer property `transaction.timeout.ms` (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 only guarantees 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.

## More...

See the Appendix in your Student Handbook for a detailed example of a transaction.

hitesh@datacouch.io

# 3: Replicating Data: A Deeper Dive



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 5 lessons:

- How Does Kafka Determine Which Messages Can be Consumed?
- How Does Kafka Place Replicas and How Can You Control Replication Further?
- How Does Kafka React When a Leader Dies?
- How Does Kafka Track Leadership Changes?
- How Does Kafka Track Follower Responsiveness?

Where this fits in:

- Hard Prerequisite: Fundamentals course
- Recommended Prerequisite: How Producers Write Messages Reliably

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Describe data replication in a Kafka cluster
- Explain how a cluster recovers from failures

hitesh@datacouch.io

# 3a: How Does Kafka Determine Which Messages Can be Consumed?

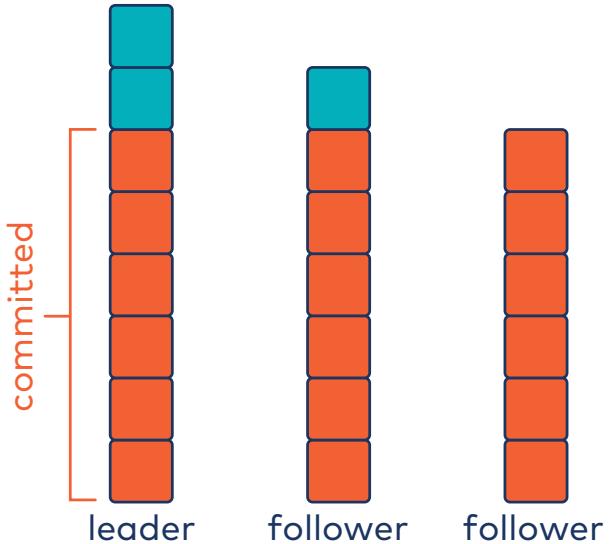
## Description

High Water Mark. Committing messages.

hitesh@datacouch.io

# Which Messages can be Consumed?

- A message is called "committed" when it is replicated by all the replicas in the ISR list
- Consumers **can only** read committed messages
- The leader decides when to commit a message
- The **High Water Mark (HWM)** is the offset of most-recently committed message
  - The High Water Mark is checkpointed to disk



A message is marked committed by the leader if all the in-sync replicas have fetched the message successfully. A committed message is guaranteed to have the same offset number on all 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.

## Refining **ISR** Definition

- We said before that a follower that has all the messages the leader also has is considered an **in-sync replica**
- More accurately, a replica is an **in-sync replica** if it has all the messages the leader has *up to and including the high water mark*. So ...
  - a leader may have messages that are not committed
  - a follower that does not have some or all not-committed messages is still an in-sync replica

hitesh@datacouch.io

# 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.

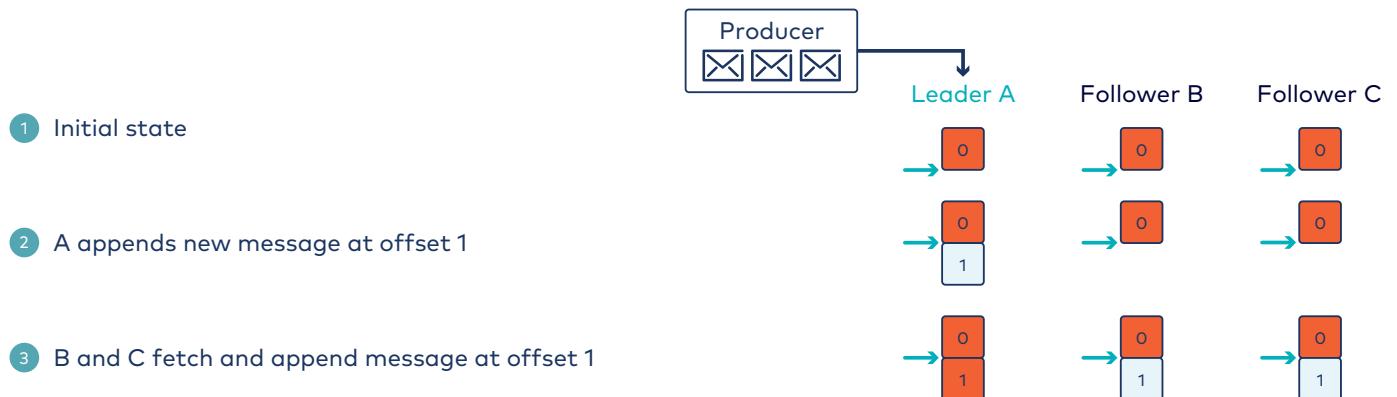
But how does the leader know when the other replicas have received the message? Traditional networking would use **ack** messages. However, that would add significant load to our networks if every replica is sending an **ack** for every message it receives. Kafka will use a more elegant approach that 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.

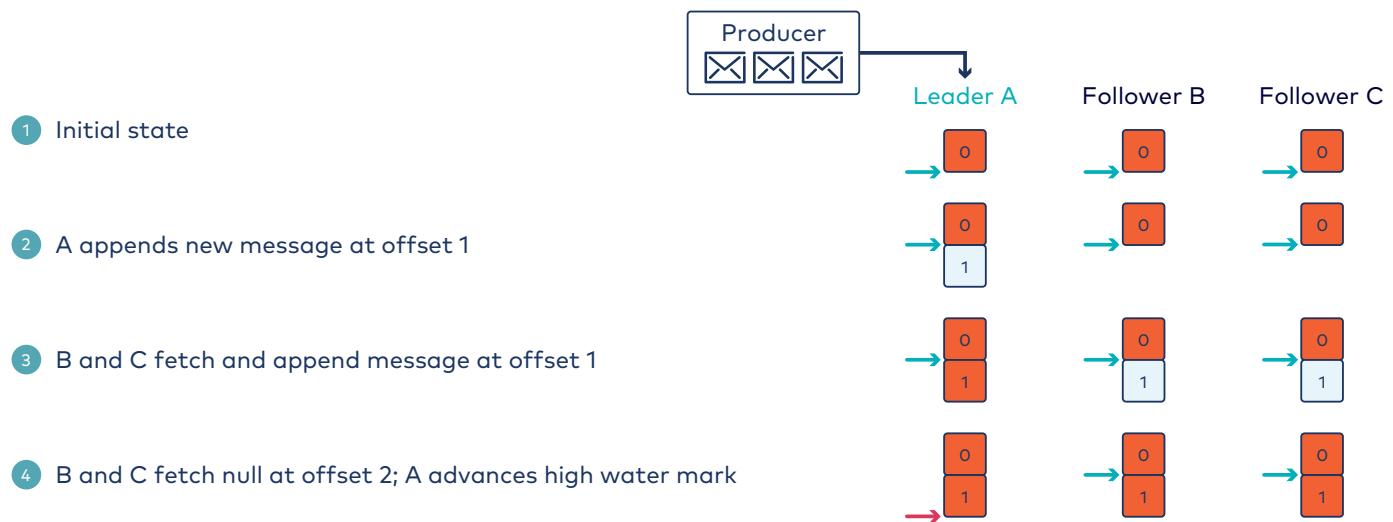
# Committing Messages with Replicas (3)



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.

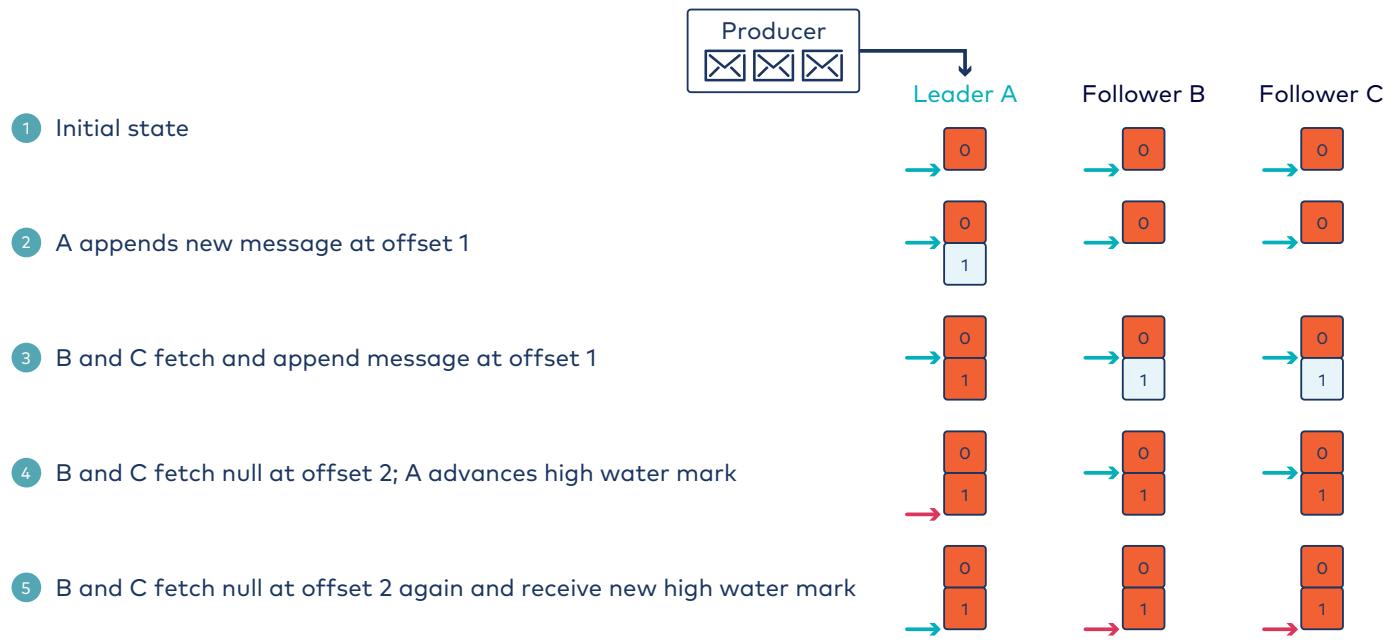
# Committing Messages with Replicas (4)



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 time out 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.

# Activity: Assessing Consumption of a Recent Message

Discuss:

**Scenario:**

- Partition  $p$ 's leader got a message at time 7.
- Consumer  $c$ , configured correctly and assigned to partition  $p$  polls and gets an empty object back at time 8.

How is this possible? Use appropriate Kafka terminology.



hitesh@datacouch.io

---

# 3b: How Does Kafka Place Replicas and How Can You Control Replication Further?

## Description

Replica placement. Preferred replicas. Under-replicated and offline partitions.

hitesh@datacouch.io

# Replica Placement

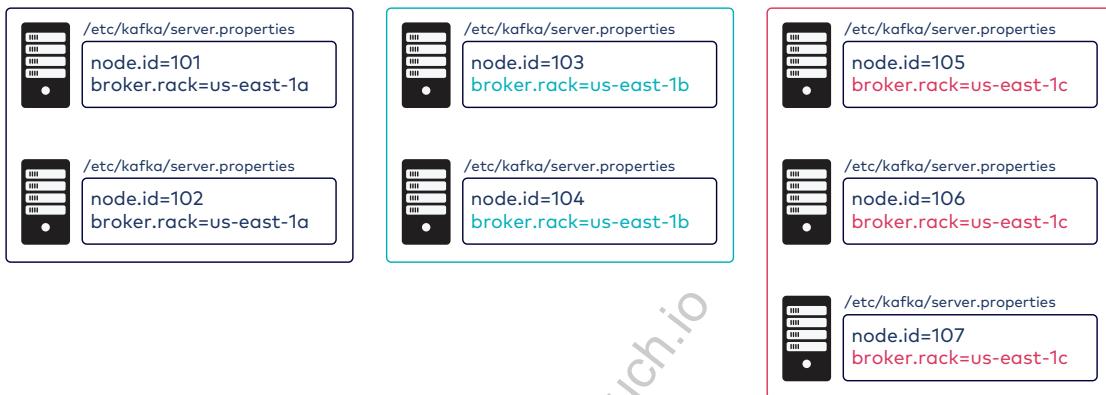
Kafka...

- places replicas on brokers
- tries to balance the placement of replicas across brokers

hitesh@datacouch.io

# Rack Awareness

- Configure `broker.rack` in `server.properties`
- 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**



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-premises 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).

# Balancing Leadership and Preferred Replicas

Kafka attempts to balance brokers in terms of how many leaders are on one broker

For each partition...

- a broker is designated as the ideal place for it - the *preferred replica*
- failover can move where the leader is
- Kafka monitors how many leaders are on the ideal broker
  - When a threshold is exceeded, leaders are moved

hitesh@datacouch.io

# Viewing Partition Placement Across Cluster (1)

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
```

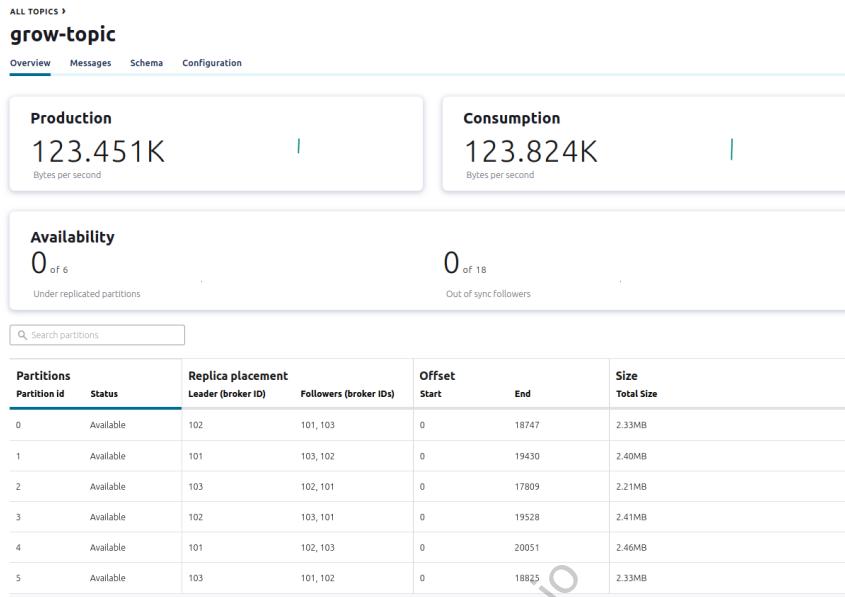


Preferred replicas highlighted in bold

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.

# Viewing Partition Placement Across Cluster (2)

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 shows where replicas are placed and whether they are out of sync.

A broker usually contains both leaders and followers, depending on which partition you are talking about. For example, broker 102 in the image contains the leader for partitions 1 and 4, and contains followers 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.

# Two More Important Definitions

## **Under-Replicated Partition**

partition where the number of in-sync replicas < replication factor

## **Offline partition**

partition for which no leader exists

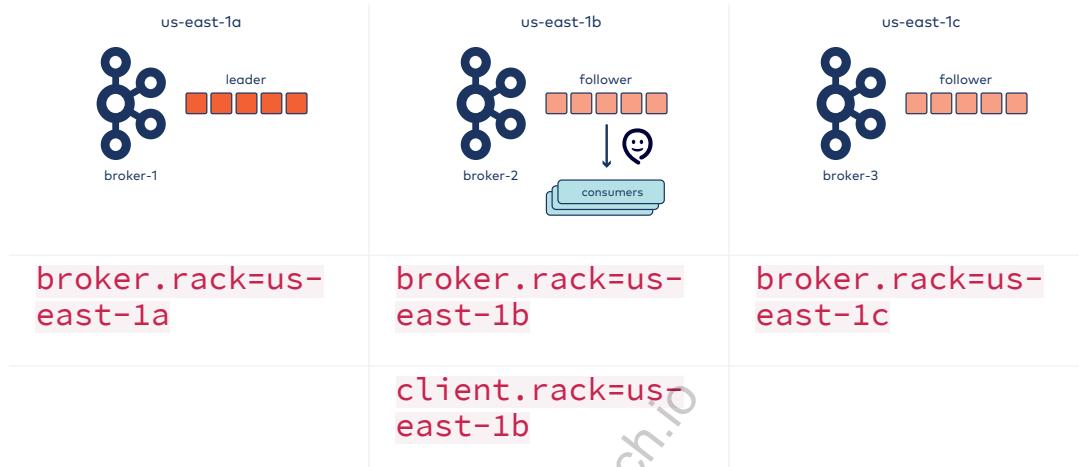


You can monitor both of these. More on that in the appendix.



# Follower Fetching

In this course, we told you all clients must interact with the leader. But, it is possible to configure Kafka so consumers fetch from followers in the same "rack" to reduce costs...



All nodes:

```
replica.selector.class=org.apache.kafka.common.replica.RackAwareReplicaSelector
```

**Question:** What is the tradeoff?

Answer to question: Followers follow the leader and fetch records from the leader periodically. Thus, if we fetch from a follower, we might not have the most up-to-date information. But the follower will catch up. The cost is increased latency.

Follower fetching was introduced in AK 2.4 with the `client.rack` property for consumers. Brokers must have rack awareness configured with `broker.rack` and `replica.selector.class` so that clients can discern where the closest replicas are.

You may wonder whether producers also have a `client.rack` setting. They do not. Kafka's fault tolerance and strong ordering guarantees are due to the append-only nature of the log and the fact that producers write only to the leader. If producers could write to followers, then the leader and follower logs would diverge.

For more information, see [KIP-392](#).

# Activity: Analyzing Replication Factor & Partition Status



Suppose we have the following replica placement - and all replicas are in their respective ISR lists:

broker 101	broker 102	broker 103	broker 104
$p_{0,L}$	$p_{1,L}$	$p_{2,L}$	$p_{1,F0}$
$p_{2,F0}$	$p_{0,F0}$	$p_{1,F1}$	$p_{2,F1}$

Then:

1. What can you deduce about replication factors from the figure?
2. How many topics are in play?
3. Suppose broker 102 goes down. Normal behavior happens. What new designations apply to any (which) partition(s)?
4. Suppose 102 is still down and 101 goes down too. What new designations apply to any (which) partition(s) now?

# 3c: How Does Kafka React When a Leader Dies?

## Description

Active Controller. Leader election. Unclean leader election.

hitesh@datacouch.io

# The Active Controller

- Kafka needs to maintain internal metadata (such as ISRs). This is stored in the system **metadata** topic
- The leader of the metadata topic is called the Controller Leader, **Active Controller** or just Controller

The Active Controller also:

- Monitors that brokers are alive
- Facilitates leader election
- Persists partition state to metadata topic
- Pushes leadership/ISR changes to involved brokers

---

There is a chicken-and-egg problem here: the Active Controller is a leader, but it is also the responsible for choosing leaders. Kafka solves this with KRaft. We will discuss the role of the Active Controller in more depth later, as well as breaking this dependency cycle.

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 managed by the Kafka Nodes configured with the **controller** role, instead of the more typical **broker** role.

The Active Controller monitors the health of every other node through keep-alive messages.

## Offline Partition or ...

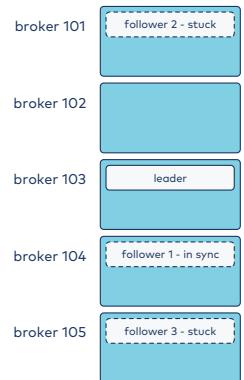
Consider the scenario at the right.

1. Suppose broker 103 goes down.

We know Kafka will automatically do leader election and pick a follower from the ISR to become the leader.

2. Suppose broker 103 goes down and later broker 104 also goes down.

Now what?



hitesh@datacouch.io

## Unclean Leader

- Kafka defaults to choosing a follower from the ISR
  - What if you want to allow followers that aren't in the ISR to become the leader?
    - 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`)
- 

`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.

# 3d: How Does Kafka Track Leadership Changes?

## Description

Leader epoch. Broker recovery.

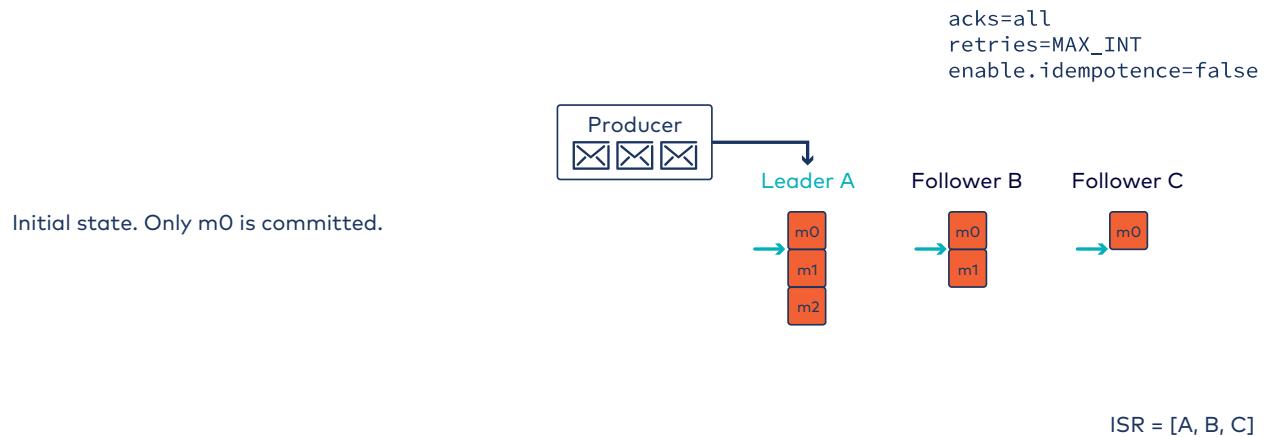
hitesh@datacouch.io

# 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
- Starts at **0**, increments
- Checkpointed to disk in **leader-epoch-checkpoint**
- What is stored:
  - The most recent epoch
  - lines of pairs: leader epoch, offset

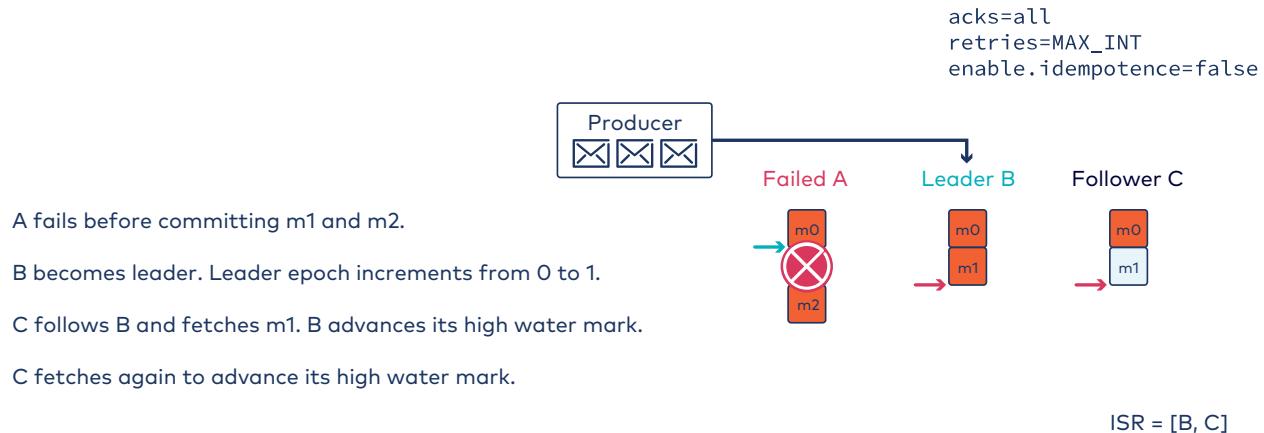
hitesh@datacouch.io

# 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)



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`, then broker A only acknowledges to the Producer that message m0 was successful. Since broker A failed before the message replicated to the follower replicas, the broker did not acknowledge success to the producer for messages m1 and m2. From the producer's perspective, if you get an error (or a timeout) you must resend, but that could produce duplicates.



If the follower on 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, the replica on broker C.

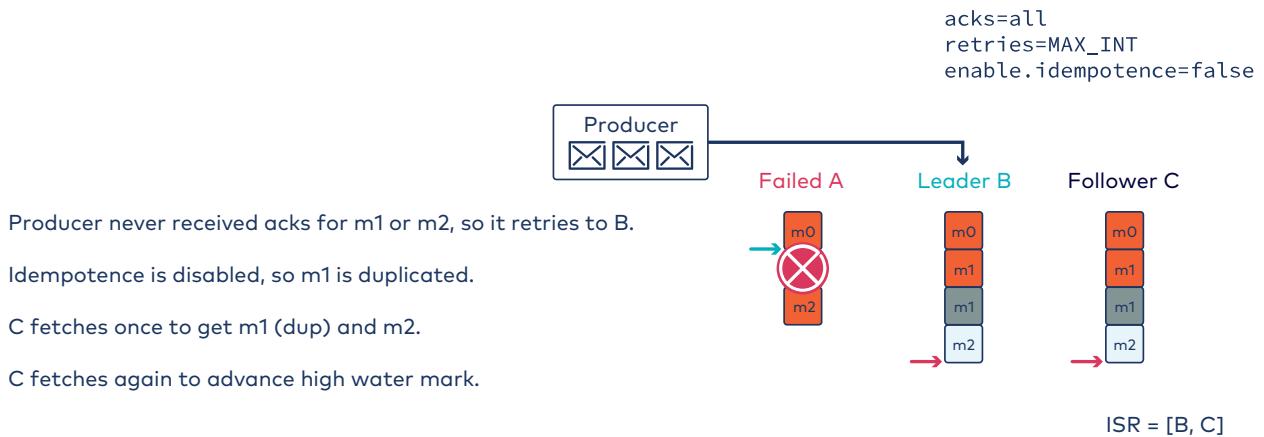
Why wasn't message one lost?:

- 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)

hitesh@datacouch.io

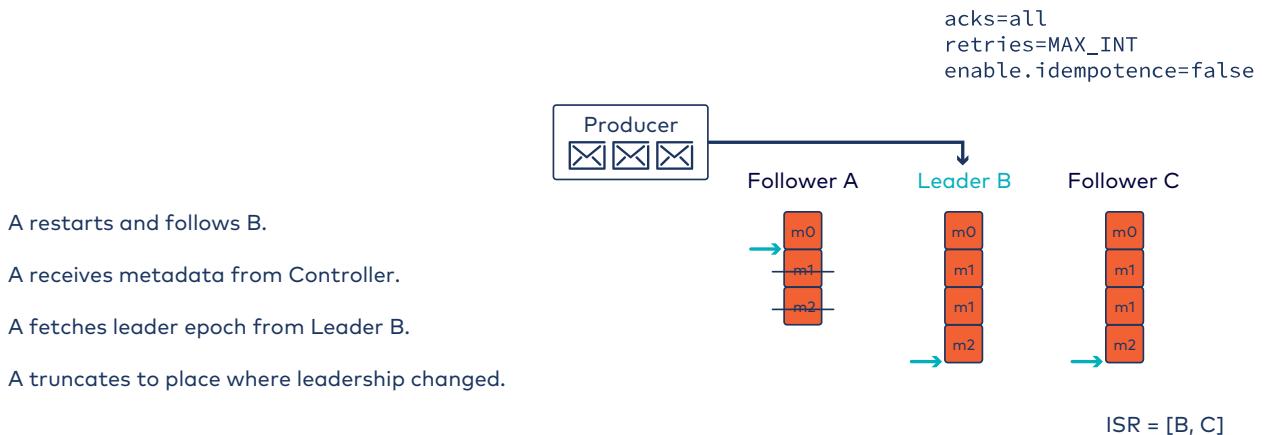
## Example of Replica Recovery (3)



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 the replica on broker B became the new leader; both m1 and m2 would be lost if the replica on broker C became the leader. This illustrates the "at most once" delivery guarantee of `acks=1`.

## Example of Replica Recovery (4)

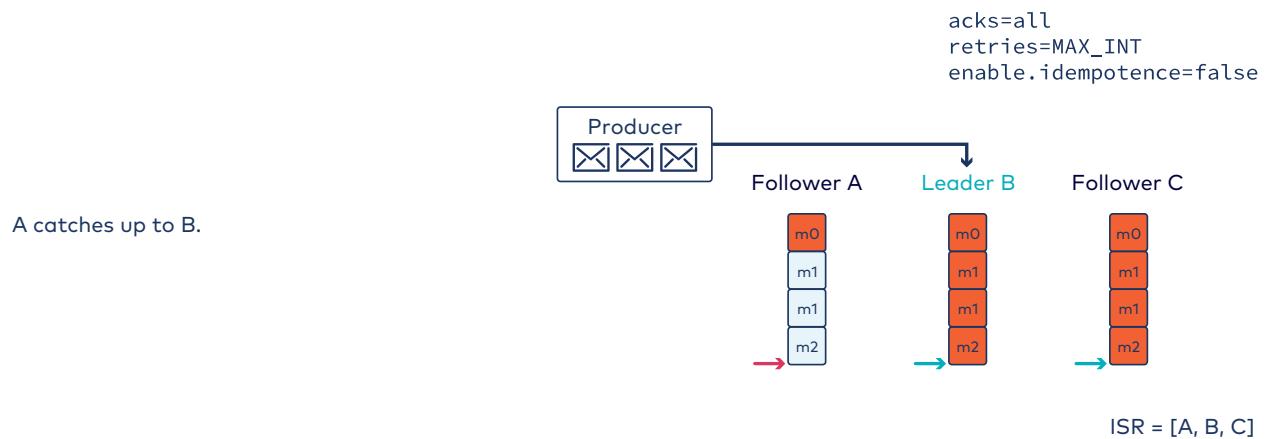


When a broker restarts, it needs to catch up to the current state of the leader. The recovering broker must ensure its log is identical to the log of the current leader. In particular, there may be some messages in this replica's log that were never committed and need to be cleaned up. The restarted broker connects to the Controller to replicate missing updates to 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](#).

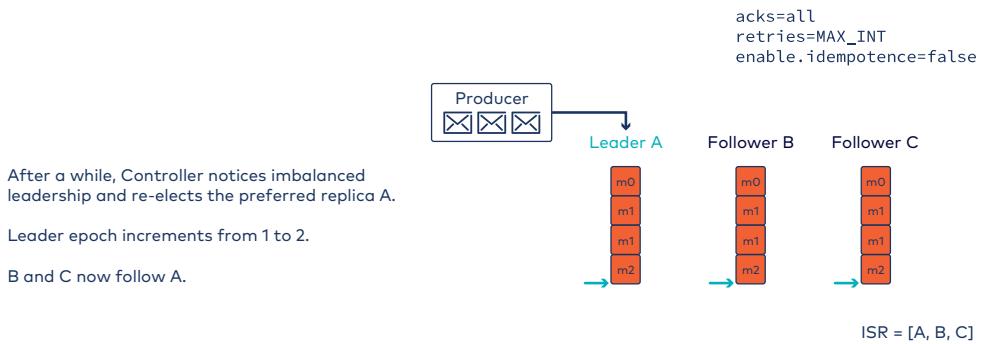
For more details about how exactly the Controller handles broker recovery, see [this source code](#). Especially note the [onBrokerStartup](#) function.

## 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)



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 minutes) 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, which is why the ISR can be restored with broker A holding the preferred replica.

# Activity: Understanding Leader Epoch



## Scenario:

For some partition  $p$ , new at time  $t = 0$ , the following timeline of events happens:

1. At  $t = 1$ , 4 messages are consumed from  $p$
2. At  $t = 3$ ,  $p$ 's follower 2 becomes  $p$ 's leader, and no other messages were written to  $p$  in the interim
3. At  $t = 4$ , 6 messages are consumed from  $p$
4. At  $t = 7$ ,  $p$ 's follower 1 becomes  $p$ 's leader

At this point, what is  $p$ 's leader epoch?

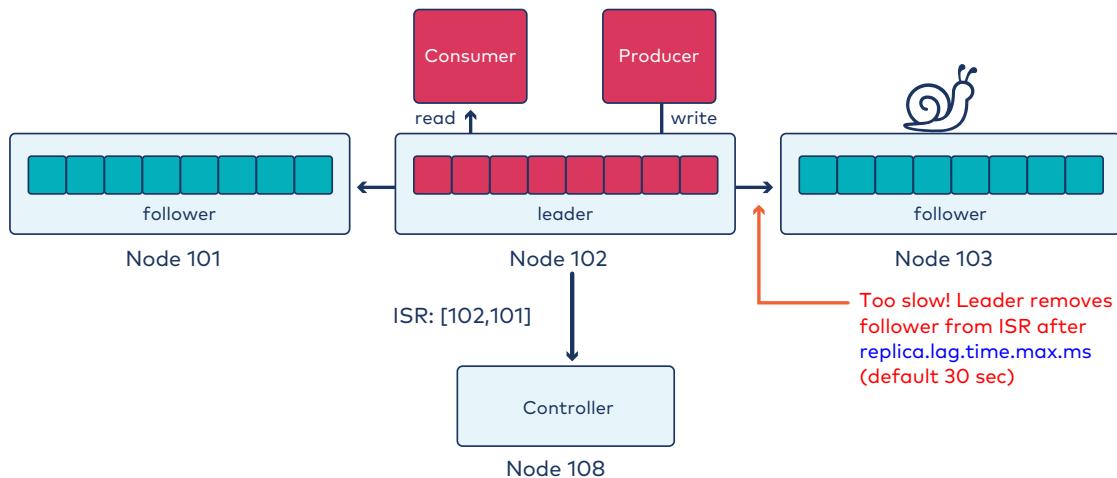
# 3e: How Does Kafka Track Follower Responsiveness?

## Description

Slow Replicas

hitesh@datacouch.io

# 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 produce request
  - Too small → replicas drop in and out of ISR
- Leader sends ISR changes to Controller
- When out-of-sync follower catches back up (by requesting the offset of a record still not produced), the leader will add back to the ISR by sending the change to the Controller.

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.

# Activity: Tuning Replication Settings



## Discussion

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

hitesh@datacouch.io

# 4: Providing Durability in Other Ways



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 3 lessons:

- How Does Kafka Organize Files to Store Partition Data?
- What are the Basics of Scaling Consumption?
- How Does Kafka Maintain Consumer Offsets?

Where this fits in:

- Hard Prerequisite: Fundamentals course
- Recommended Prerequisite: Replication & Replica Placement

hitesh@datacouch.io

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Explain how offset management impacts the end user application
- Understand how Kafka's internal offsets topic is organized

hitesh@datacouch.io

# 4a: How Does Kafka Organize Files to Store Partition Data?

## Description

Logs vs. segments. Details of files created per partition and per segment. Rolling of segments.

hitesh@datacouch.io

# 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](#).

# 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

- Additional per log segment for *certain producers*:

.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 `.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
0000000000000283423.index
0000000000000283423.timeindex
0000000000000283423.log
...
00000000000008296402.index
00000000000008296402.timeindex
00000000000008296402.log
leader-epoch-checkpoint
```

---

Segment files are named for the first offset tracked by that set of files. In the example, the files named `0000000000000283423.*` manage messages in offsets from 0000000000000283423 to 00000000000008296401 ([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.

# Log Segment Properties

- Each `.log` filename is equal to the offset of the first message it contains, e.g.,
  - `00000000000049288237.log`
- Messages are written to the **active** segment
- The active segment **rolls** to a new segment file if any are exceeded:
  - `log.segment.bytes` (Default: 1GB)
  - `log.roll.ms` (Default: 168 hours = 1 week)
  - `log.index.size.max.bytes` (Default: 10MB)
- A former active segment is called an **inactive** segment after it has rolled over

---

All 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.

---

- Separately control when to roll the segment file for the offsets topic:
  - `offsets.topic.segment.bytes`

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** (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

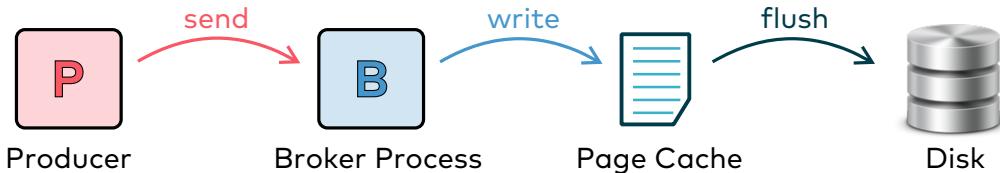
---

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



# Page Cache and Flushing to Disk



- Messages are written to partitions
- Partitions are made up of **segment files** (new segment every 1 GB by default)
- Asynchronous IO means OS writes first to the in-memory **page cache** for performance
  - Kafka consumer fetch requests benefit from **zero-copy transfer**
- Page cache is **flushed to disk**:
  - 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 a 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.

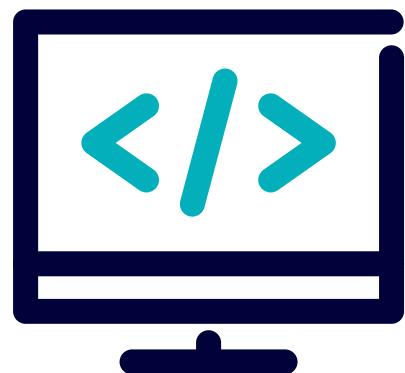
When users look at the `*.log` files, it shows data that is both flushed to disk and still in the page cache (OS buffer) that has not yet been flushed. There are linux tools (e.g. `vmtouch`) that show what has and hasn't been flushed.

Lastly, when there is a segment roll, the new inactive segment is flushed.

# Lab: Investigating the Distributed Log

Please work on **Lab 4a: Investigating the Distributed Log**

Refer to the Exercise Guide



hitesh@datacouch.io

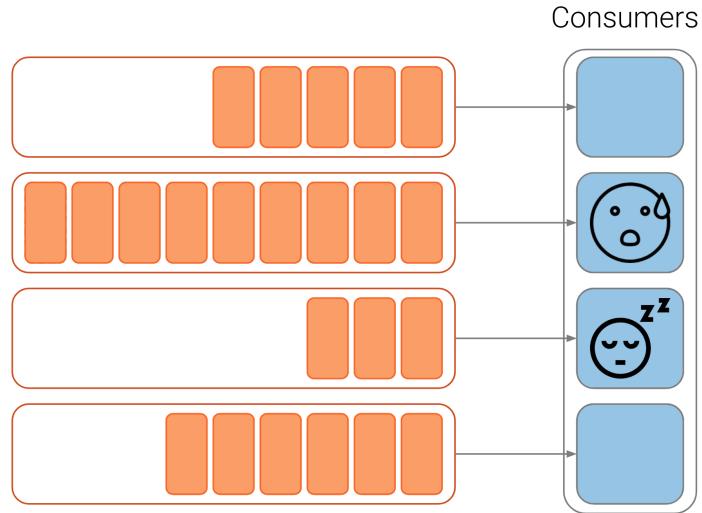
# 4b: What are the Basics of Scaling Consumption?

## Description

Consuming with one consumer vs. multiple consumers in a group vs. multiple groups.

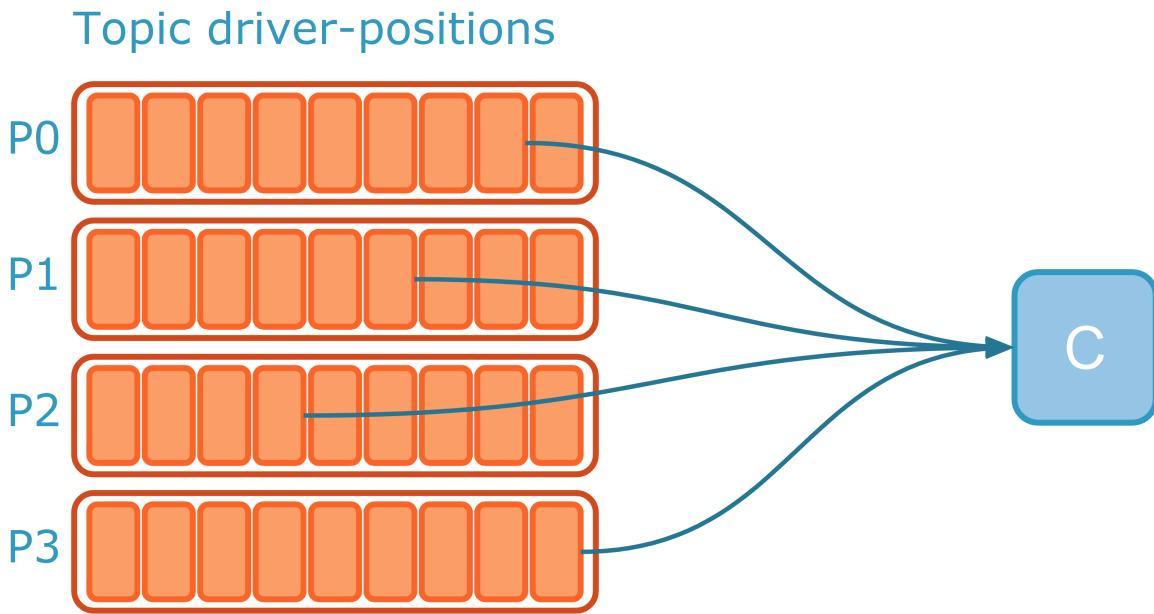
hitesh@datacouch.io

# Cardinality



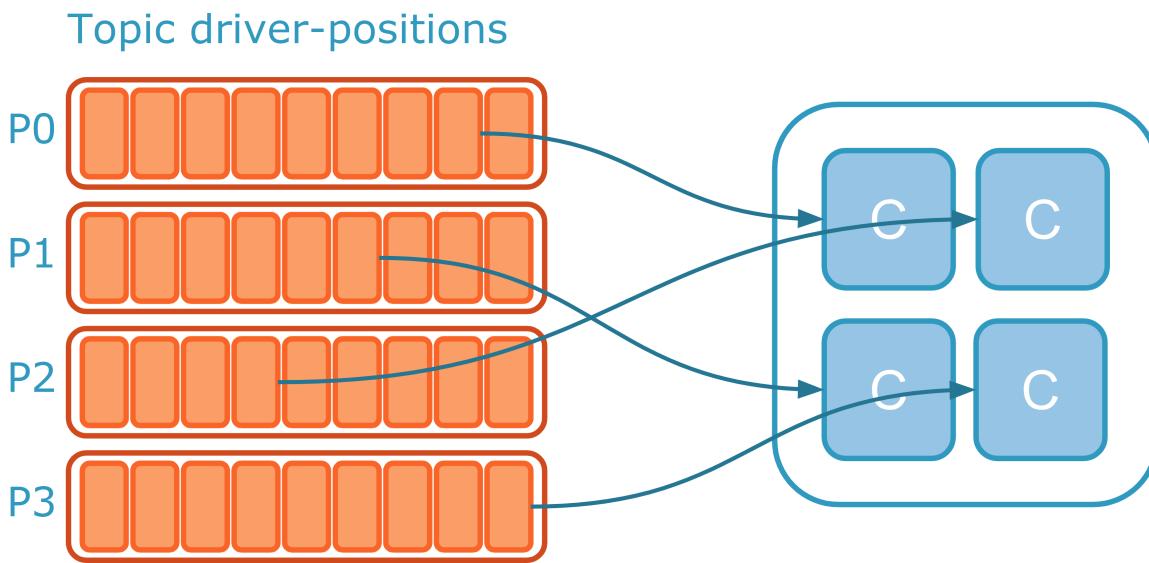
- **Cardinality:** the number of elements in a set or other grouping, as a property of that grouping.
- Key cardinality affects the amount of work done by the individual Consumers in a group. Poor key choice can lead to uneven workloads.
- Keys in Kafka don't have to be simple types like Integer, String, 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.

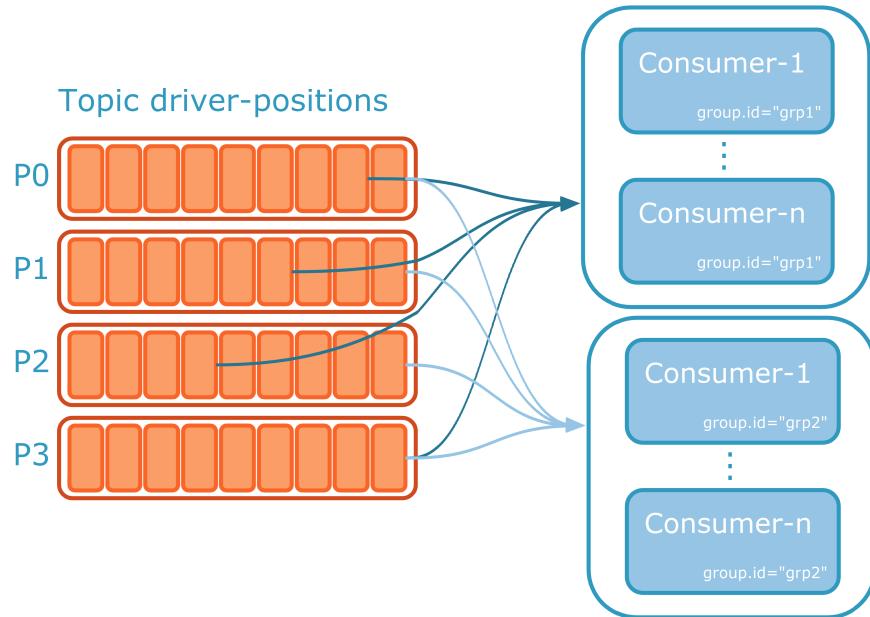
# Consuming as a Group



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).

# 4c: How Does Kafka Maintain Consumer Offsets?

## Description

Consumer offsets topic: uses, special properties. Viewing offsets.

hitesh@datacouch.io

# Consumer Offset Management

- Consumption is tracked per topic-partition
  - Each consumer maintains an offset in its memory for each partition it is reading
  - As a 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.

# 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)

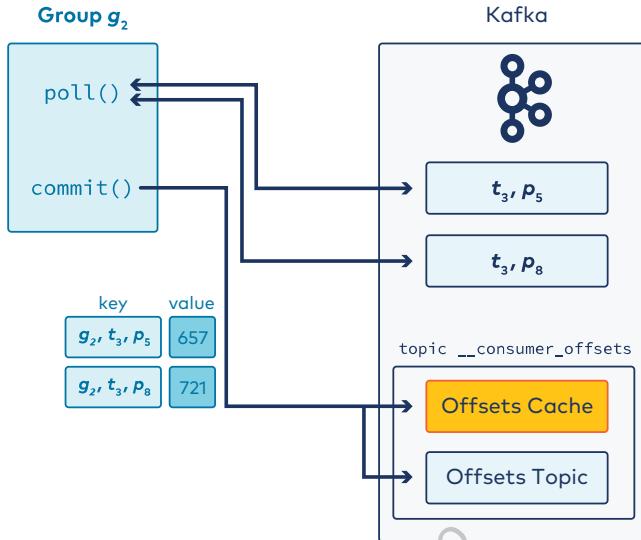


If there aren't enough brokers, auto-creation of `__consumer_offsets` fails. Consumers should only begin consuming after all brokers are running.

The setting `offsets.topic.replication.factor` will be enforced during auto topic creation for the offsets topic. If the number of brokers is fewer (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.

# Consumers and Offsets (Kafka Topic Storage)



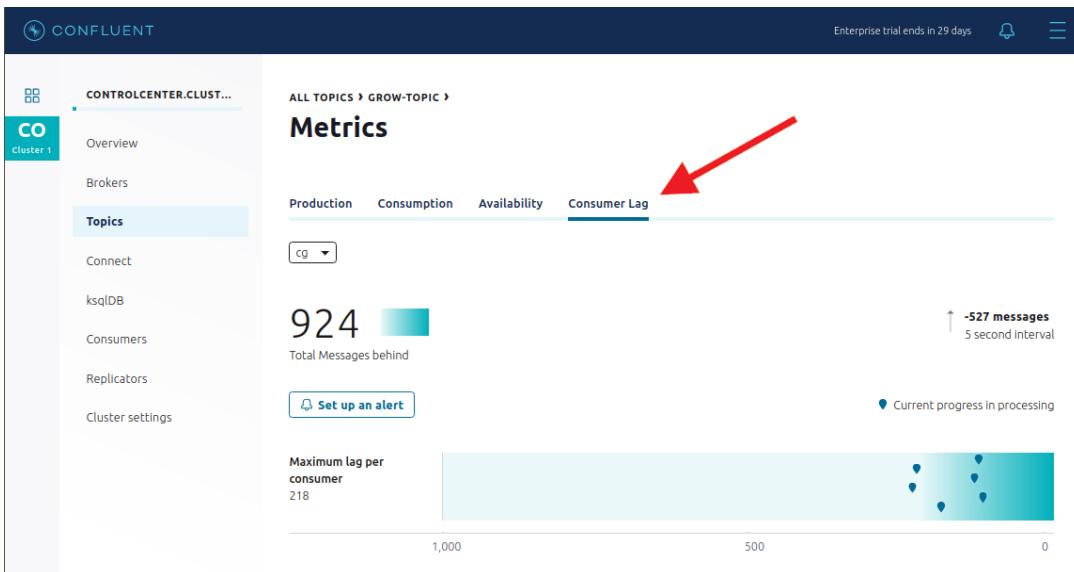
When a Consumer commits its offsets to the offsets topic, it sends four data points:

- 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 reads 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.

# 5: Configuring a Kafka Cluster



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 2 lessons:

- How Do You Configure Kafka Nodes?
- What if You Want to Adjust Settings Dynamically or Topic-wise?

Where this fits in:

- Hard Prerequisite: Fundamentals course
- Recommended Prerequisites: Other Ways Kafka Provides Durability

hitesh@datacouch.io

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Describe the basics of how to configure Kafka
- Understand static vs. dynamic configurations
- Understand topic vs. node-level configurations

hitesh@datacouch.io

# 5a: How Do You Configure Brokers?

## Description

Static node settings. Where to set. Examples of classes of node settings.

hitesh@datacouch.io

# Where to Set Node Settings?

- Properties file: `/etc/kafka/server.properties`
  - On each node
- File is only read when node (re-)starts
- Includes configuration for:
  - Node itself
  - Cluster Defaults
  - Topics Defaults



It is possible to override these node settings. More on that in the next lesson.

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

Confluent Control Center (a.k.a. CCC or C3) provides a dashboard to inspect node configuration.

Any change to `server.properties` requires a reboot of the broker to take effect.



It used to be the case that all Kafka **Servers** were **Brokers** but with AK 3.0 and KRaft, there are also **Controllers**. Furthermore, the concept of a **Node** also appears here (with `node.id`), meaning that the naming is currently in-flux. Every Kafka **Server** is a **Node**, it may be a **Broker** or a **Controller**, but in some contexts **broker** may still have the same meaning as **node**.

# Some Node Properties

Some properties are clearly "this Node" things, e.g.,

- `node.id`
  - `log.dirs` - where data directories for logs are
  - `num.network.threads` - how many network threads
  - `num.io.threads` - how many worker/IO threads
- 

These are some properties that apply to the Node where they are configured.

hitesh@datacouch.io

## Another "This Node" Config: Listeners

First off, recall, we use `host:port` pairs to describe connections, e.g., `broker1:9092`

Two node settings guide what ports are used and their security mechanism:

- `listeners` - host and port information from the local (server) endpoint
- `advertised.listeners` - host and port information from the remote (client) endpoint

You'd specify a listener something like `listeners = [protocol]://[host]:[port]`

So you might have settings like these (different examples):

- `listeners = PLAINTEXT://broker1:9092`
- `listeners = PLAINTEXT://broker-1.intranet:9092, SSL://broker1:9093`
- `advertised.listeners = SSL://broker1:9093`

---

This is another example of a pair of properties that apply to a single broker.

We will revisit this concept in more detail in the security module.

Regarding `listeners`:

- specifying the host as `0.0.0.0` will bind to all local IPv4 interfaces e.g. `listeners = PLAINTEXT://0.0.0.0:9092`
- leaving the host empty will bind to the default interface e.g. `listeners = PLAINTEXT://:9092` (which happens to be the default value)

# Cluster Defaults

Some properties you set in `server.properties` are meant as cluster defaults for all nodes.

Examples:

- `auto.create.topics.enable`



But `server.properties` is for **this one** node. If the cluster defaults are not consistent across the nodes in the cluster, you may experience unpredictable behavior.

You set properties for each node in `server.properties`. On the previous slides, we looked at properties that apply only to the current node. On this slide, we give some properties that are meant to be cluster defaults that apply to all nodes. While the `server.properties` file applies to a single node, this kind of property must be set the same on all nodes for proper behavior.

# Some Properties Related to Topics

Some node properties apply to topics:

- `default.replication.factor`
- `num.partitions`
- log rolling threshold: `log.segment.bytes` and `log.roll.ms`



Some of these can be overridden at the topic level.

---

These two properties are configured in `server.properties` but they apply to topics.

We learned about log rolling in the last module.

What's special about these properties is that it is possible to have topic overrides that take precedence over these settings. We go into that mechanism in the next lesson.

# What if I Don't Configure a Property?

Kafka defaults apply!

See documentation.

But, there's more. See next lesson...

hitesh@datacouch.io

# Example Config File

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

```
# 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
node.id=[An integer]
listeners=[list of listeners]
auto.create.topics.enable=false
min.insync.replicas=2
```

For more configuration information, see

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

# 5b: What if You Want to Adjust Settings Dynamically or Apply at the Topic Level?

## Description

Dynamic vs. static node settings. Cluster-wide vs. per-broker settings. Topic overrides. Order of precedence.

hitesh@datacouch.io

# There Aren't Only Static Node Settings...

Other options:

- Dynamic broker settings
  - Dynamic cluster-wide settings
  - Topic overrides
- 

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 the cluster metadata topic.

- Because static configuration is stored in files, it is easy to manage them using version control systems or directly managed by 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 non-replicated topics), but there isn't a predefined way to control versions or that CI/CD pipelines leverage this mechanism

# Order of Precedence

1. Topic settings
  2. Dynamic per-broker config
  3. Dynamic cluster-wide default config
  4. Static server config in `server.properties`
  5. Kafka default
- 

As we expand beyond only static configurations, consider this order of precedence for all the levels of configurations available.

Note that dynamic changes will be stored in the cluster metadata topic.

hitesh@datacouch.io

# Viewing Dynamic Broker Configurations

Display dynamic broker configurations for broker with ID `103`:

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



To display all config settings, not just dynamic changes, specify `--all`.

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`.

# Changing Broker Configurations Dynamically

- Change a cluster-wide default configuration

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

- Change a broker configuration

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



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

# Deleting a Broker Config

Delete a broker configuration

```
$ kafka-configs \
--bootstrap-server kafka-1:9092 \
--broker 107 \
--alter \
--delete-config min.insync.replicas
```



**Question:** What will the value of `min.insync.replicas` be now?

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

# Topic Overrides

Topic level configurations to override broker defaults



The names are often different, but similar.

Examples:

Meaning	Topic Override	Broker Config
Threshold log segment size for rolling active segment	<code>segment.bytes</code>	<code>log.segment.bytes</code>
Maximum size of a message	<code>max.message.bytes</code>	<code>message.max.bytes</code>

There are several properties that can be set at the broker level and apply to all logs managed by that broker. Additionally, many of these have topic-level overrides that can be set to take precedence over broker-level settings. Put differently, you may want to set a configuration to apply to all partitions of a topic, regardless of what brokers those partitions live on. Topic-level overrides allow for that.

# 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
```

hitesh@datacouch.io

## 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.

# 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 ([documentation](#)) 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.



## 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
- Command:

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

# Activity: Troubleshooting Configuration Confusion



## Discussion:

- A colleague insists he changed the rollover time for the active segment to 2 hours.
- But another colleague is reporting that she has seen some log segments for topic  $t_7$ , partition  $p_{12}$  are on broker 103 have been the active log segment with timestamps 4 and 5 hours in the past.

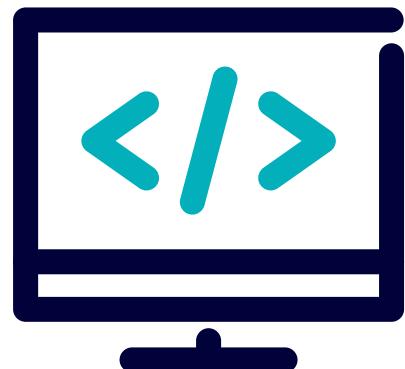
Another colleague wants answers and explanations. What do you tell them?

hitesh@datacouch.io

# Lab: Exploring Configuration

Please work on **Lab 5a: Exploring Configuration**

Refer to the Exercise Guide

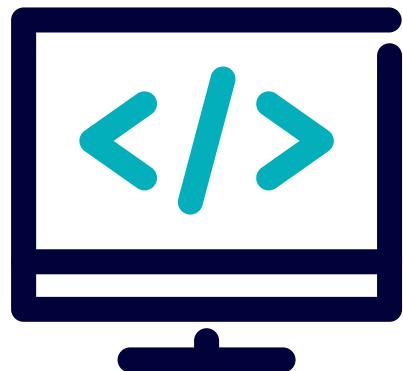


hitesh@datacouch.io

# Lab: Increasing Replication Factor

Please work on **Lab 5b: Increasing Replication Factor**

Refer to the Exercise Guide



hitesh@datacouch.io

# 6: Managing a Kafka Cluster



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 6 lessons:

- What Should You Consider When Installing and Upgrading Kafka?
- What is a Controller vs a Broker?
- What are the Basics of Monitoring Kafka?
- How Can You Decide How Kafka Keeps Messages?
- How Can You Move Partitions To New Brokers Easily?
- What Should You Consider When Shrinking a Cluster?

Where this fits in:

- Hard Prerequisite: Fundamentals course
- Recommended Prerequisite: Other Ways Kafka Provides Durability

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- 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

hitesh@datacouch.io

# 6a: What Should You Consider When Installing and Upgrading Kafka?

## Description

Considerations for installation and upgrading.

hitesh@datacouch.io

# Methods of Installation

- You can deploy Confluent Platform from
  - A Tar archive
  - DEB or RPM package
  - Docker container



Schema Registry needs to have Kafka installed first

- Deploy in a distributed environment using one of...
  - Confluent for Kubernetes (formerly called Confluent Operator)
  - Ansible playbooks

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.

For reference, a list of services:

- Apache Kafka®
- Confluent Schema Registry
- REST Proxy
- Confluent Control Center
- Kafka Connect (distributed mode)
- ksqlDB
- Confluent Replicator

# Local vs. Distributed Installation

Local	Distributed
<ul style="list-style-type: none"><li>• When installed on a single machine.</li><li>• Installed via zip, tar, or Docker images</li></ul>	<ul style="list-style-type: none"><li>• When services are installed across several machines</li><li>• Installed via Kubernetes or Ansible</li></ul>

Docker is supported on:

- Linux amd64 & x86\_64
- Linux arm64 & aarch64

Notes on distributed:

- Installs Confluent Platform using packages or archives.
- Starts services using `systemd` scripts.
- Provides variables for configuring security settings for Confluent Platform.
- Provides options for monitoring Confluent Platform

[This simple wizard](#) will help you launch a self-managed, multi-node deployment of Confluent Platform using Ansible.

- Enterprise: Zip , Tar, Docker, Kubernetes, Ansible
- Community Edition: Installed via Zip, Tar or Docker

# Upgrading a Cluster (1)

There are many things to consider when upgrading the Confluent Platform:

- The order of upgrade for Kafka nodes (controllers and brokers) and the rest of the Confluent Platform is very important.
- Always read the upgrade documentation on our website to get the full scope of what the upgrades entail.

hitesh@datacouch.io

## Upgrading a Cluster (2)

- Be aware of the broker protocol version and the log message format version between the various releases.
- Apply a license key for the Confluent Platform.
  - Know how to apply the key.
- Do rolling restart—the entire cluster stays up as you take down each broker one by one to upgrade.



See link in your guide!

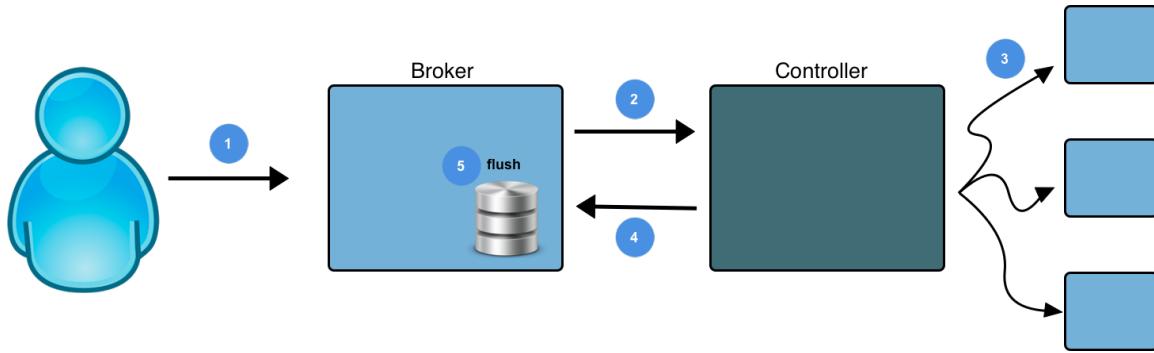
Upgrading to a new version of Kafka can require special steps if there are fundamental changes to Kafka like the message format or inter-broker protocol. To upgrade in these cases (for example, upgrading to Kafka 2.2 from any older version), one must set `inter.broker.protocol.version` and `log.message.format.version`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



Refer to [this upgrading documentation](#) for more specific instructions

If using a configuration management tool, rolling restarts must be taken into account. The CP Ansible Playbook supports graceful rolling restarts. The Confluent Kubernetes Operator automatically performs a graceful rolling restart in response to applying a change.

# Controlled Shutdown



1. Administrator sends a **SIGTERM** to the Java process running Kafka, e.g., `kafka-server-stop`
2. The node sends request to the Controller.
3. The Controller facilitates leader elections
4. The Controller responds to the node that it can now leave.
5. The node flushes file system caches to disk.
6. The node 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 Java process running Kafka, e.g., `kill -SIGTERM <pid>`
2. The node sends a request to the Controller to inform it's planning to shut down.
3. The Controller will determine which partitions the node contains a leader for and forces a leader election for each of these partitions to other nodes.
4. Once the elections are complete, the Controller acknowledges the node's shutdown request.
5. The node explicitly flushes the file system caches to disk.
6. The node 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 node 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 node contained 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 node 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.

hitesh@datacouch.io

## 6b: What is a Controller vs a Broker?

### Description

KRaft. Kafka Roles. Controller Election. Metadata Topic.

hitesh@datacouch.io

# Cluster Metadata and the Controller

## Cluster Metadata

- Stored in `--cluster_metadata` topic
- Active Controller is its Leader
- Contains:
  - Topic and Dynamic Configuration
  - Partition Leader, ISR
  - Quotas
  - ACLs
  - ...

## Active Controller

- Monitors Nodes' Health
- Controller Chooses Leaders
- High-Availability Role

Can you spot the problem?

You may have spotted a circular dependency in the second bullet points: the leader of the Metadata Topic becomes the Controller, so the Controller cannot elect itself! Kafka needs another method of electing the Controller: we will present the solution in the next slide.

The `--cluster_metadata` topic has a single partition 0, therefore a single leader. This is because being the Active Controller is the same as being the Controller Leader (of a partition of the metadata topic), and we cannot have multiple controllers at the same time.



*Controller Leader* is another name for the Active Controller. In this course we use *Active Controller* or just *Controller* to avoid confusions like "the controller leader chooses leaders" or what a controller follower is early in the course.

## KRaft

- Implementation of a Distributed Consensus Algorithm within Kafka
  - Voting system with quorum (3 or 5 nodes)
  - Quorum voters elect the leader of `__cluster_metadata`
- 

KRaft (K-Raft) is a pull-mode variant of the **Raft** algorithm developed for Kafka. When the algorithm initializes or the leader fails, the remaining quorum voters will elect a new leader among themselves (in our case, the *Controller Leader* or *Active Controller*).

To avoid split vote when all voters are up, the cluster must have either 3 or 5 quorum voters. This number will become, as a consequence, the **replication factor** of `__cluster_metadata`.

For further details on the algorithm, you can watch this [video](#).



Older version of Kafka relied on **Zookeeper** for metadata management and controller election. AK 4.0 is planned to stop supporting this mode, so you may be interested in live-migration: <https://docs.confluent.io/platform/current/installation/migrate-zk-kraft.html>

# Roles for Kafka Nodes

A Kafka cluster comprises a number of voter nodes plus some broker nodes.

**Quorum voter**    `process.roles=controller` (for `__cluster_metadata` topic)

**Broker**                `process.roles=broker` (for all the other topics)

All nodes must know how to locate the quorum voters:

`controller.quorum.voters=nodeid1@host1:port1,nodeid2@host2:port2,nodeid3@host3:port3`

---

Metadata topic management is a critical job, so a subset of nodes is dedicated just to controller election and management of this single topic.



Mixed-mode `process.roles=controller,broker` is meant for non-production only clusters, and it is unsupported

All quorum voters need to know how to contact the others. On the other hand, all brokers must have a way to find the Controller Leader, which is a quorum voter. That's why all nodes need the `controller.quorum.voters` configuration.

# Metadata Topic Creation

The following steps should be taken when creating a new cluster:

1. Install Kafka on voters and configure

```
node.id=<unique integer for each node>
process.roles=controller
quorum.controller.voters=<node.id1>@<host1>:<port1>,...
controller.listener.names=PLAINTEXT
metadata.log.dir=/var/lib/metadata_mount_point # optional
```

2. On **one** voter, generate the Cluster ID (this formats the metadata directory too)

```
$ ./bin/kafka-storage random-uuid
q1Sh-9_ISia_zwGINzRvyQ
```

3. On the remaining nodes (voters and brokers), format the metadata directory

```
$ ./bin/kafka-storage format -t q1Sh-9_ISia_zwGINzRvyQ
```

We again face a circular dependency problem: Kafka will not start with a metadata topic, then how do we create that topic?

First of all, all Kafka clusters have unique **Cluster IDs**. It is a randomly-generated UUID and the `kafka-storage` command allows you to generate one.



All nodes in the same cluster use the same Cluster ID. Different clusters **must** use different Cluster IDs

Regarding `controller.listener.names`, it should contain the list of listener ports used by the controller role. If you use advanced listener configuration (as we see in the Security module) change the configuration value accordingly. E.g.

```
listeners=CONTROLLER://:9093
listener.security.protocol.map=CONTROLLER:SSL
controller.listener.names=CONTROLLER
```

The `__cluster_metadata` topic follows the same layout on disk than all the other topics (`.log`, `.index` files etc.), but we can store its contents on a different storage device (e.g. SSD/NVMe). This may be for performance or reliability reasons. In that case, `metadata.log.dirs` points to the mount point of that device. If not configured, the topic

will be stored on the first entry in `log.dirs` (`/var/lib/kafka` by default)

Also notice that the metadata directory must be formatted on brokers (`process.roles=broker`). Brokers cache the topic locally, reducing the workload on the Controller for certain operations (e.g. ACL validation)

hitesh@datacouch.io

# 6c: What are the Basics of Monitoring Kafka?

## Description

Basics of metrics and monitoring tools. Kafka logs vs. system logs.

hitesh@datacouch.io

# 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 (events/messages/records) 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.

# Kinds of JMX Metrics

There are two kinds of JMX metrics:

- **gauge** - a measure of something *right now*
    - e.g., number of offline partitions
  - **meter** - a measure of event occurrence over a time sample
    - e.g., count, weight, throughput
- 

Various JMX metrics will come up in this course; this slide simply distinguishes between the two classes of JMX metrics.

hitesh@datacouch.io

# Configuring the Cluster for Monitoring

- Enable JMX metrics by setting `JMX_PORT` environment variable

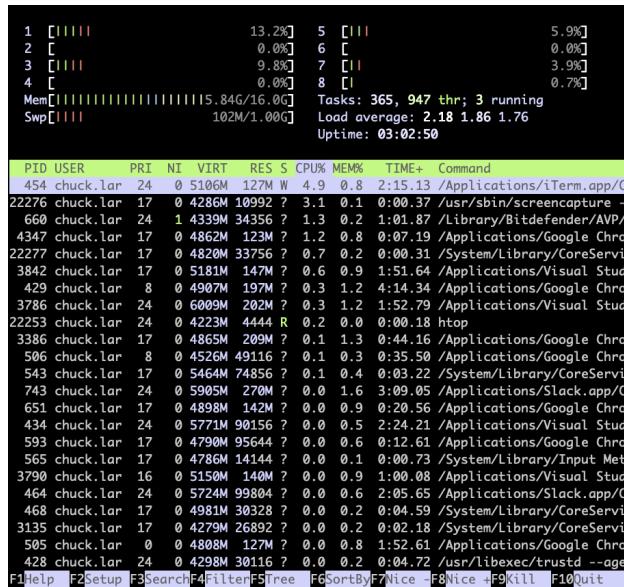
```
$ export JMX_PORT=9990
```

- 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).

# Monitoring Kafka at the OS Level



- Open file handles
  - Set `ulimit -n 100000`
  - 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. Consider:

- 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.



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.

# Troubleshooting Issues

- Check metrics
  - Parse the `log4j` logs
  - Avoid unnecessary restarts
- 
- If needed, enable more detailed level of logging in `log4j.properties` (e.g. `WARN` → `TRACE`)
    - Alternatively, logging priority can be changed as a dynamic node configuration with `kafka-configs`
  - 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](#).

# 6d: How Can You Decide How Kafka Keeps Messages?

## Description

Deletion. Compaction. Examples. Details of implementation of compaction. Monitoring and logging of compaction.

hitesh@datacouch.io

# Managing Log File Growth

- `cleanup.policy` at topic level
  - `delete` (default)
  - `compact`
  - both: `delete,compact`



Regardless of the policy, the active segment is kept intact as it is still open in read+append mode.

Messages 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 node 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.

The default policy is configured in `server.properties` by setting `log.cleanup.policy`.

*hitesh@datacouch.io*

# 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 superseded 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.

Here is a handy list of topic overrides to customize the behavior at topic level.

<code>server.properties</code>	<code>Topic override</code>
<code>log.roll.ms</code>	<code>segment.ms</code>
<code>log.segment.bytes</code>	<code>segment.bytes</code>
<code>log.retention.ms</code>	<code>retention.ms</code>
<code>log.retention.bytes</code>	<code>retention.bytes</code>

# Deleting All Messages in a Topic

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)
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 **most recent** value for a given key

- Examples:
    - Database change capture
    - Real-time table look-ups during stream processing
    - Maintaining a topic of temperatures per postal code
    - Tracking the progress of e-commerce orders
- 

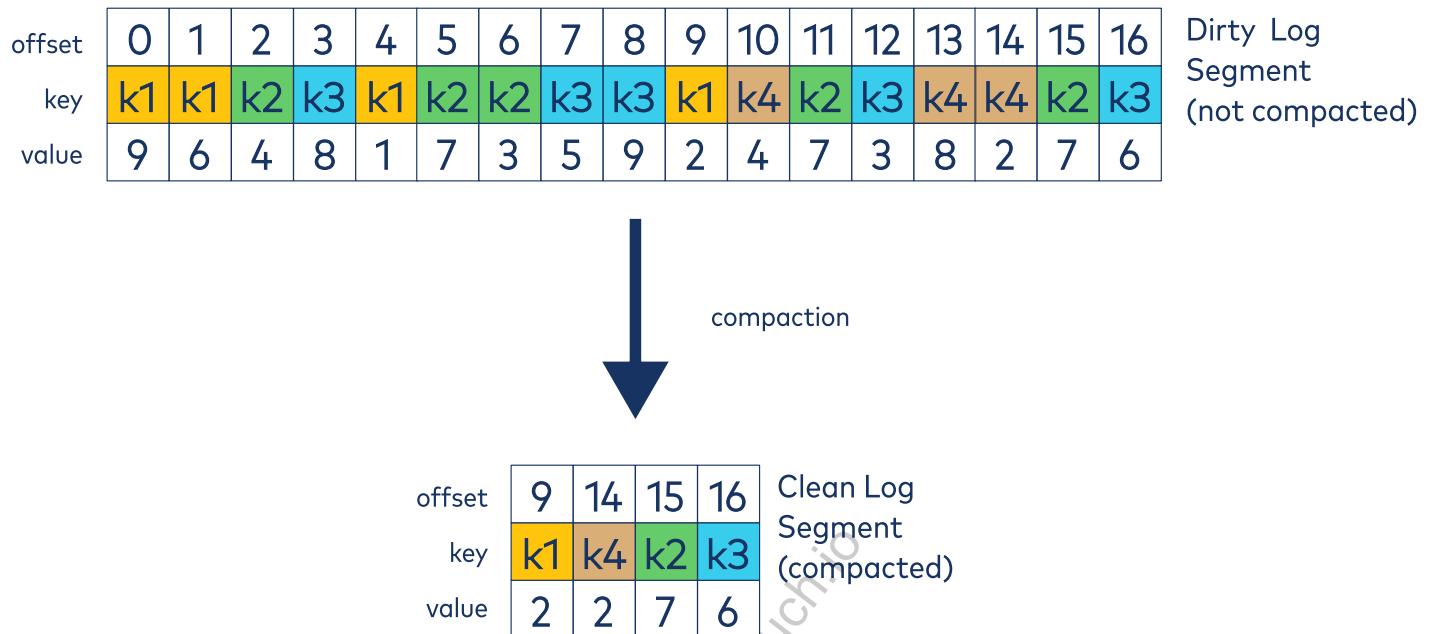
The **compact** cleanup policy is specific to keyed messages. It is designed for applications that require only the freshest value of any given key, for example to help in-memory applications recover from failure with the latest state.

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?

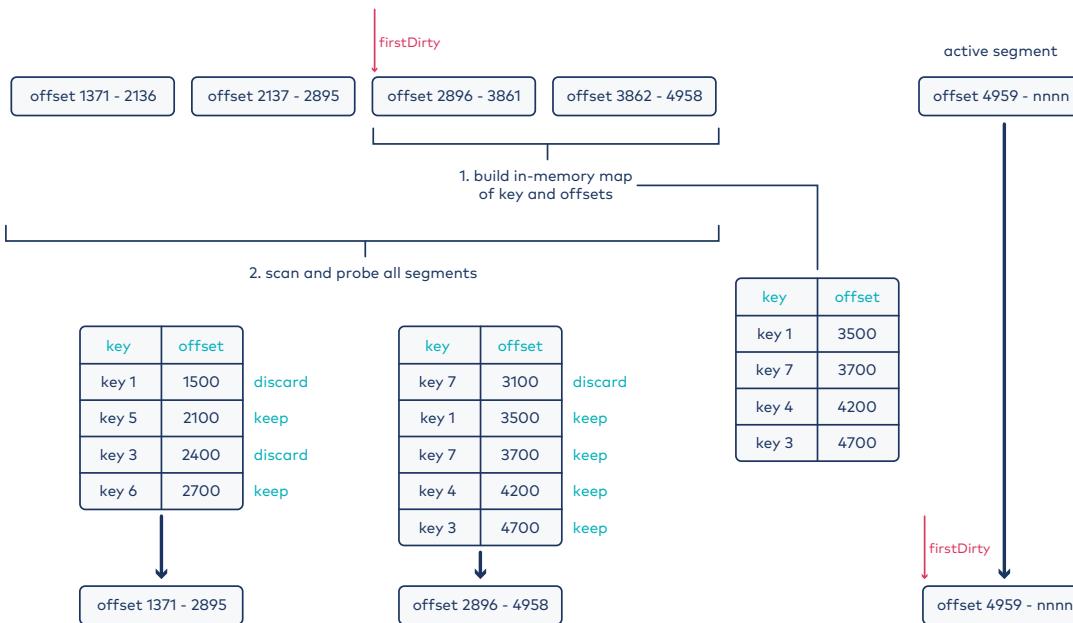


Notice here that we keep only the latest value for each key.

Notice also that after compaction, messages retain their offsets. In other words, offsets will likely be non-contiguous after compaction.

Back to that idea that we can't delete individual messages? Aren't we doing that here? It sure looks like it. However, as per implementation details beyond the scope of this course, when compaction runs, it creates new copies of log segments with only the messages that are kept. Then it moves file pointers around and deletes old segments as a whole. It does not alter the existing segments directly.

# 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.



## Deleting Keys with Log Compaction

Compaction yields a log where there is at most one instance of any key in the inactive segments.

But what if you don't want a key at all anymore?



- 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 SLO for delete retention (`delete.retention.ms`). This SLO 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.

# 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.min.compaction.lag.ms` (default 0)  
The minimum time a message will remain uncompacted in the log
- `log.cleaner.max.compaction.lag.ms` (default infinite)  
The maximum time a message will remain ineligible for compaction
- `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 many 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`.

If the `log.cleaner.max.compaction.lag.ms` or the

`log.cleaner.min.compaction.lag.ms` configurations are also specified, then the log compactor considers the log eligible for compaction as soon as either: (i) the dirty ratio threshold has been met and the log has had dirty (uncompacted) records for at least the `log.cleaner.min.compaction.lag.ms` duration, or (ii) if the log has had dirty (uncompacted) records for at most the `log.cleaner.max.compaction.lag.ms` period.

hitesh@datacouch.io

# 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.

# 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.

# 6e: How Can You Move Partitions To New Brokers Easily?

## Description

Basics of Auto Data Balancer.

hitesh@datacouch.io

# Auto Data Balancer Overview

- Applies when:
  - You've added new (empty) brokers and want to move partitions to them
  - You've noticed disk usage among brokers is not balanced
- What it does:
  - Calculates an optimal placement of partitions among brokers
  - Automatically moves partitions
- Paid Confluent feature

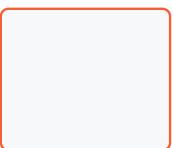
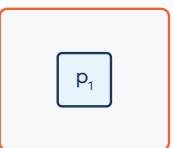


See also: Self-Balancing Clusters

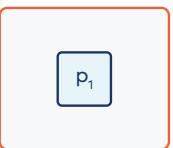
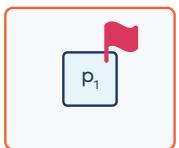
# Partition Movement with Auto Data Balancer



ISR list: [0, 1]



Add new replica of  $p_1$  on new broker  
ISR list: [0, 1, 2]



Switch leader of  $p_1$  over to 2 & delete original replica  
ISR list: [2, 1]



Disk utilization increases during rebalance

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 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 cannot 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	<code>kafka-reassign-partitions</code>
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 stats	No
Decommission brokers	Yes, automated with <code>--remove-broker-ids</code>	Yes, manual reassignment
Increase replication factor	No	Yes, with manual work
Balance partitions across log dirs	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 you are interested in this tool, look at 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.

# 6f: What Should You Consider When Shrinking a Cluster?

## Description

Tips for shrinking a cluster. Replacing a server.

hitesh@datacouch.io

# 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.

# Replacing a Failed Node

- On new hardware/container, deploy a new Kafka Server with the same value for `node.id`
- The new Server will automatically bootstrap data on start-up
- If possible, start up the replacement Server at an off-peak time



Server 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 Server is:

1. Replace the hardware
2. Install the OS and Kafka
3. Assign the Server the identity of the lost system (`node.id` and `controller.quorum.voters` plus other configuration)
4. Start the Server and let the replication features rebuild the Partitions.

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

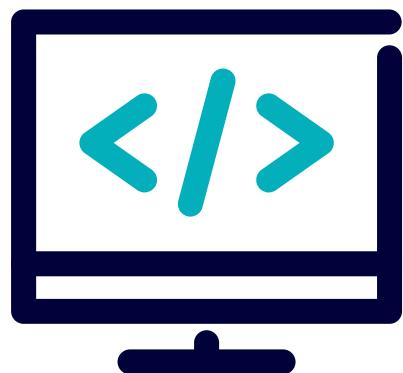
Be careful—the Server will copy the data as fast as it can during recovery and there is no throttling tool. This can have a significant impact on the network. If possible, it is recommended to avoid scenarios where a broker loses all of its data and has to recover every partition via replication. For 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 [this reference](#) for more details.

# Lab: Kafka Administrative Tools

Please work on **Lab 6a: Kafka Administrative Tools**

Refer to the Exercise Guide



hitesh@datacouch.io

# 7: Balancing Load with Consumer Groups and Partitions



CONFLUENT  
**Global Education**

# Module Overview



This module contains 3 lessons:

- How Do Partitions and Consumers Scale?
- How Do Groups Distribute Work Across Partitions?
- How Does Kafka Manage Groups?

Where this fits in:

- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite: Other Ways Kafka Provides Durability

hitesh@datacouch.io

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Explain how consumer groups read messages from Kafka
- Explain the relationship between partitions and throughput

hitesh@datacouch.io

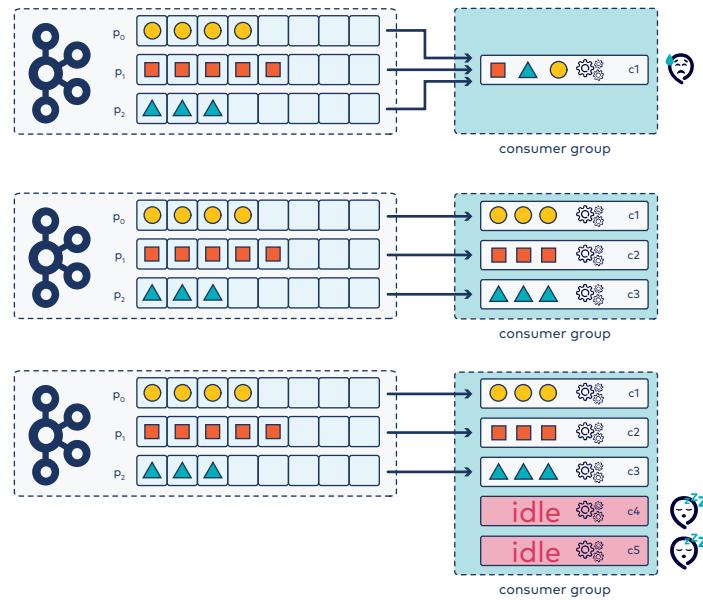
# 7a: How Do Partitions and Consumers Scale?

## Description

Scalability of consumer groups. Adding partitions. Benefits and challenges of adding partitions.

hitesh@datacouch.io

# 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 10 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, e.g.:

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

Notes:

- 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

---

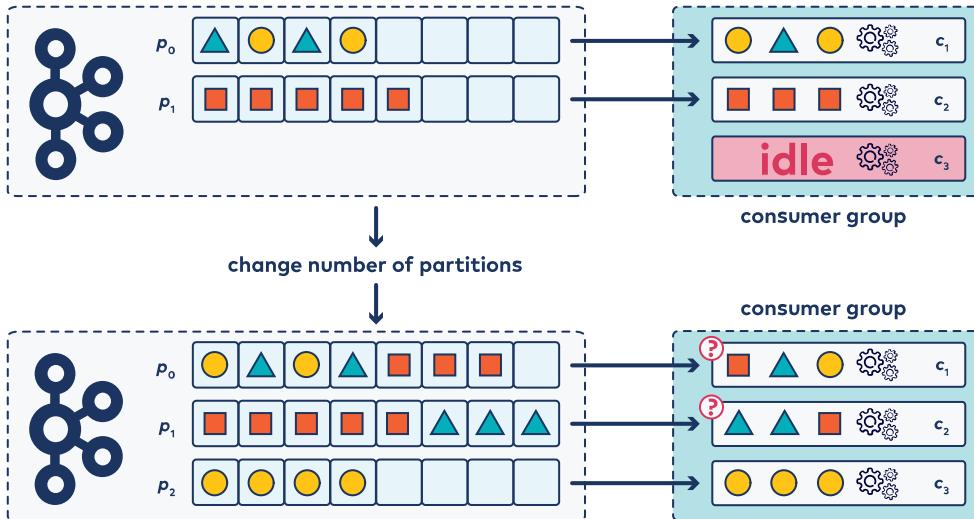
Remember that key-based partitioning uses `hash(key) mod <number-of-partitions>`, so increasing partition count means what used to go to, e.g., partition 0 might now go to partition 15. So messages with a given key will no longer be on a single Partition.

For the workaround (new topic, copy data), you will have to update all 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 partitioning 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.



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...

# Number of Partitions

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

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.

Another lesson and lab will address how to measure throughput on a per-producer and per-consumer level.

Other considerations:

- 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`)

# 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 soft limits shown on the slide (<4K partitions/broker, <200 K partitions per cluster) were calculated on AK2.0 and Zookeeper: we don't have updated information for KRaft-based clusters yet. In any case, most environments are well below these values (typically in the 1000-1500 range or less per broker).

# 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 a Controller election.
  - 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.

# 7b: How Do Groups Distribute Work Across Partitions?

## Description

Assignment of partitions to consumers. Strategies: range, round-robin, sticky, cooperative sticky.

hitesh@datacouch.io

# 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

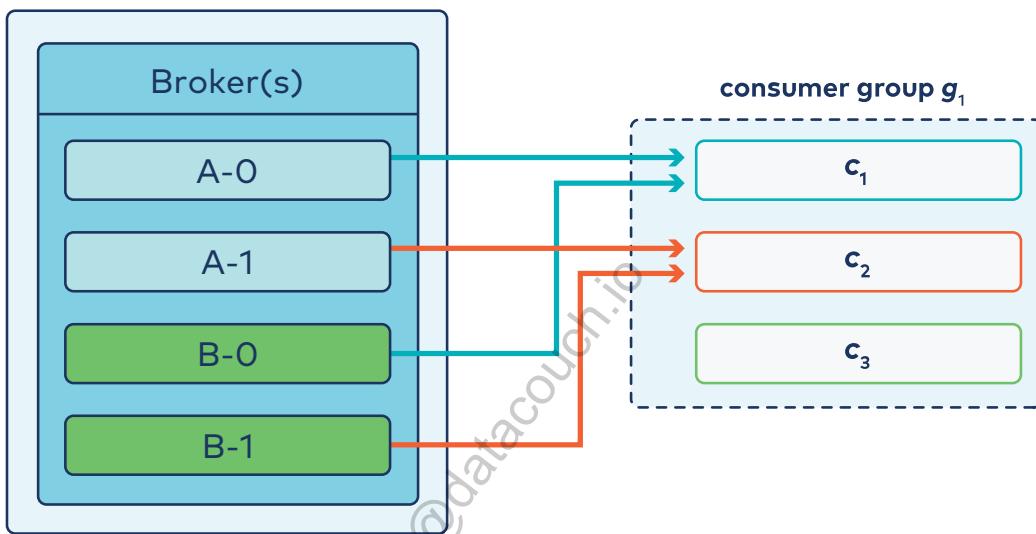
---

The consumer group is managed by a process called a group coordinator.

hitesh@datacouch.io

# Partition Assignment Strategy: Range

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

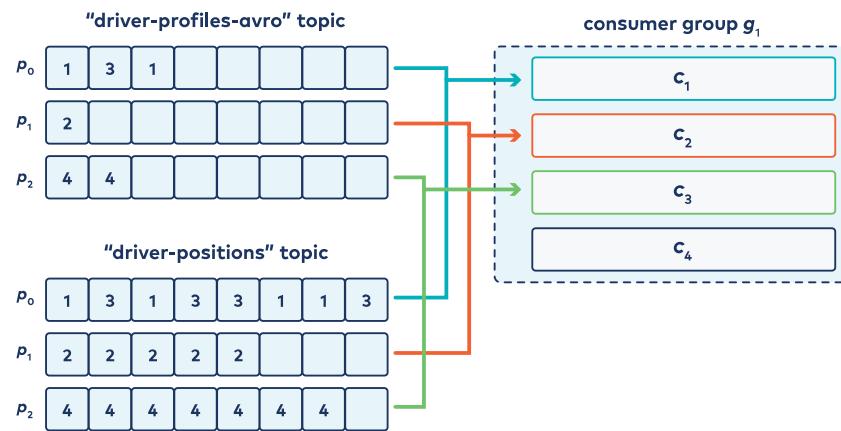


There are four built-in partition assignment strategies: Range (default), RoundRobin, Sticky and CooperativeSticky. These are set with `partition.assignment.strategy` in the consumer code.

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 user ids; Topic B is tracking search clicks for the same set of user ids. 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.

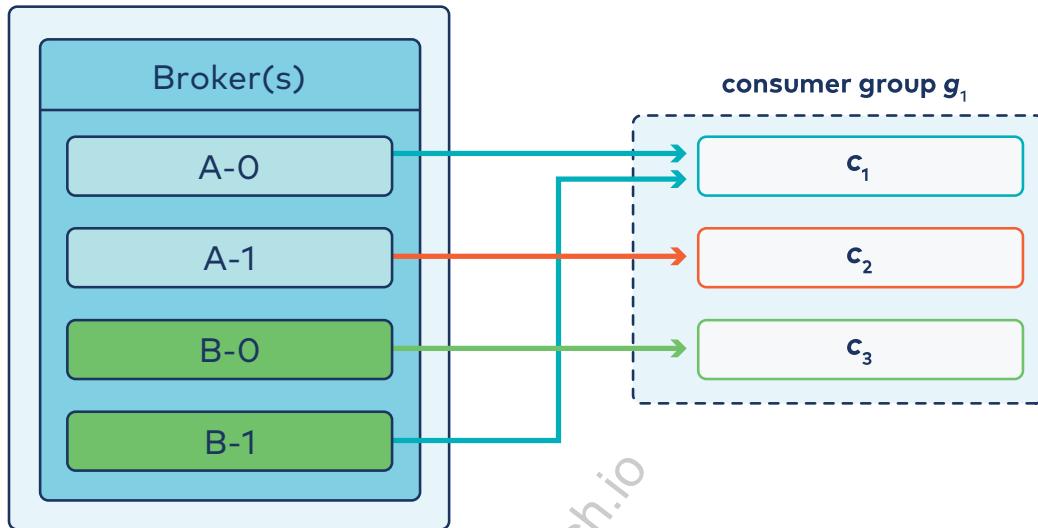
Here's a more complex version of the diagram showing keys and more intended for a development audience where we get into needing to keep same-key messages together:



hitesh@datacouch.io

# 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 [KIP 54](#) for a full description of this feature.

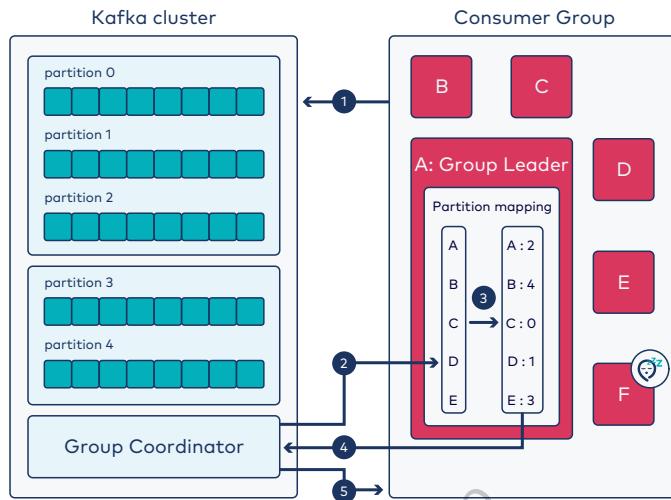
- CooperativeSticky

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 [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 process running on a broker), and the group leader (a process running on 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 broker containing the leader for this partition is selected to run 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 run the group leader process. 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 one 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 the cluster metadata.
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 running the group coordinator for a given `group.id` with:

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

## 7c: How Does Kafka Manage Groups?

### Description

Group management. Rebalances. Heartbeats and failure detection.

hitesh@datacouch.io

# 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



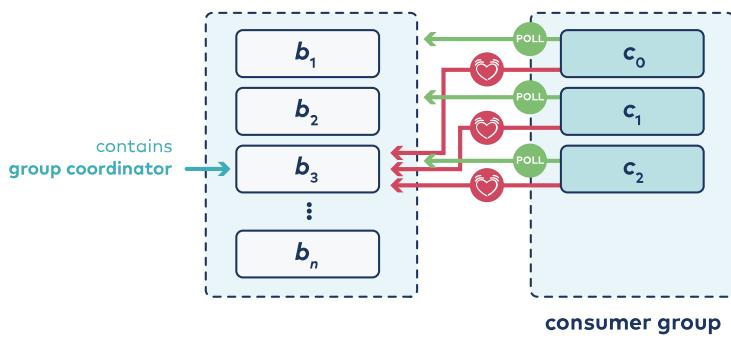
Consumption pauses during rebalance. Avoid unnecessary rebalances.

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 s)
- `session.timeout.ms` (Default: 45 s)
  - If no heartbeat is received in this time, consumer is dropped from 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`
- Rebalances are still triggered on heartbeat and poll timeouts



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/>.

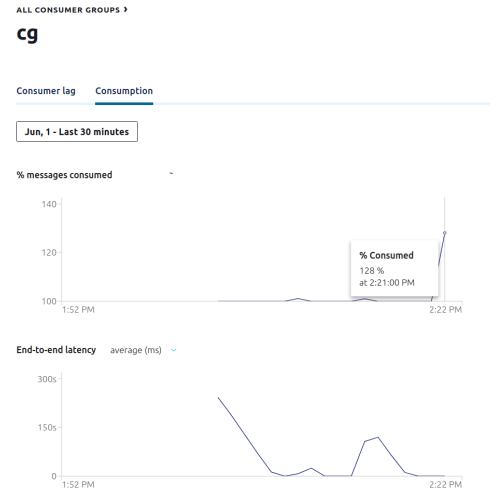
Most features of static group membership were released in Apache Kafka 2.3. Kafka Connect uses a version of it for cooperative task rebalancing. See:

- <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

- 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.

- `--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\$OffsetMessageFormatter"  
...  
[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

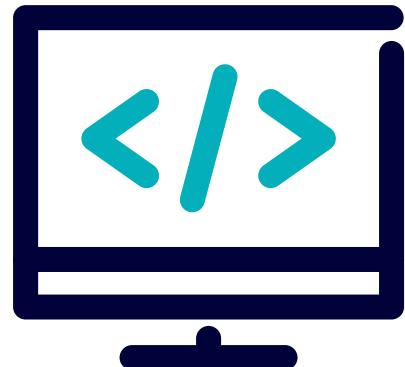
plus this retention period.

*hitesh@datacouch.io*

# Lab: Modifying Partitions and Viewing Offsets

Please work on **Lab 7a: Modifying Partitions and Viewing Offsets**

Refer to the Exercise Guide



hitesh@datacouch.io

# 8: Optimizing Kafka's Performance



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 7 lessons:

- How Does Kafka Handle the Idea of Sending Many Messages at Once?
- How Do Produce and Fetch Requests Get Processed on a Broker?
- How Can You Measure and Control How Requests Make It Through a Broker?
- What Else Can Affect Broker Performance?
- How Do You Control It So One Client Does Not Dominate the Broker Resources?
- What Should You Consider in Assessing Client Performance?
- How Can You Test How Clients Perform?

Where this fits in:

- Hard Prerequisite: Managing a Kafka Cluster

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

- Explain why batching helps performance
- Describe the anatomy of requests on a broker
- Performance-tune the cluster

hitesh@datacouch.io

# 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 (MB/s). Even a small three-broker cluster with modest hardware can achieve on the order of 10-100 MB/s. 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.

# 8a: How Does Kafka Handle the Idea of Sending Many Messages at Once?

## Description

Batching. Pipelining. Tuning batching. Compression.

hitesh@datacouch.io

# 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:

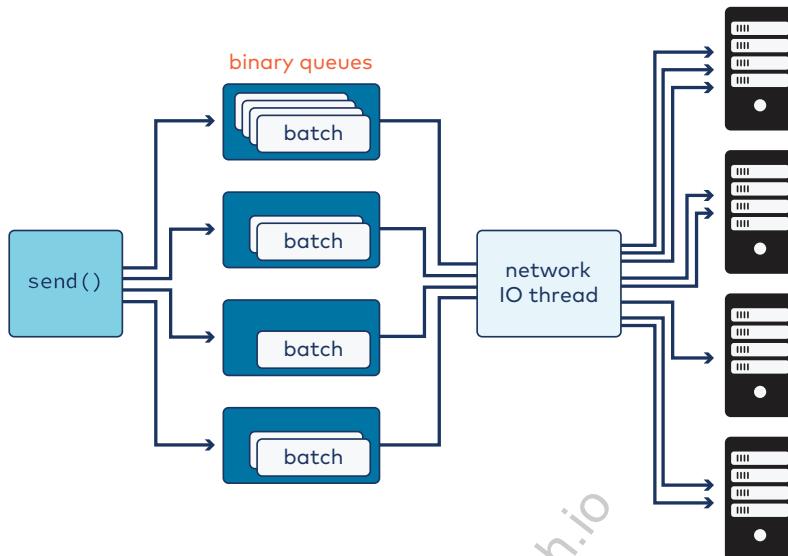
1. There is a steady flow of SmartCars. The first student enters the SmartCar and goes to school at the speed limit. After that car leaves, there is another SmartCar that the second student takes, and so on.
2. One big bus comes to pick up all the students at once before heading to school at the speed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

# Producer Architecture

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



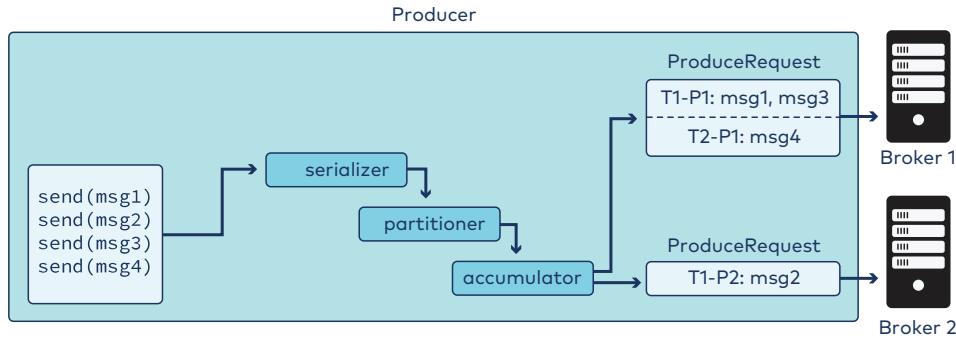
When a producer creates a message, pushing that data to the cluster is a two-stage process:

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 broker containing the leader 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)

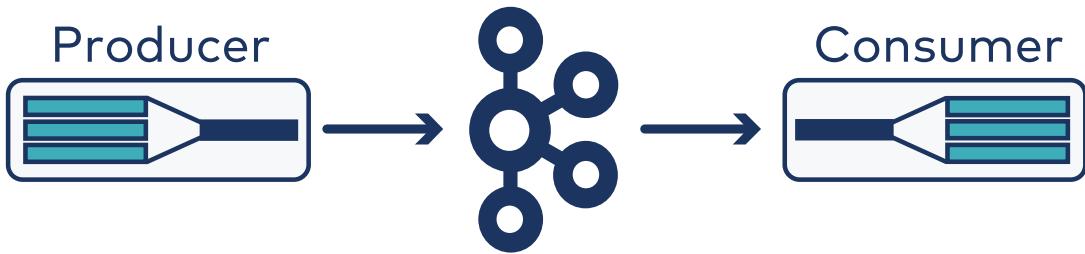
- `batch.size` (Default: 16 KB):
  - The maximum size of a batch before sending
- `linger.ms` (Default: 0, i.e., send immediately):
  - Time to wait for messages to batch together

---

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.

hitesh@datacouch.io

# End-To-End Batch Compression



1. Producer batches messages and compresses the batch
2. Compressed batch stored in Kafka
3. Consumer decompresses and un-batches messages

---

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 and Latency

- `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: 32 MB
  - 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 16 KB batch size, the minimum size for the buffer is 160 KB. 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 resources on producers. Producers typically have enough CPU and memory for compression, so usually it is best to leave broker and topic `compression.type` settings to `producer`.



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

# Tuning Consumer Throughput and Latency

- High throughput:
  - Large `fetch.min.bytes` (Default: 1)
  - Reasonable `fetch.max.wait.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.

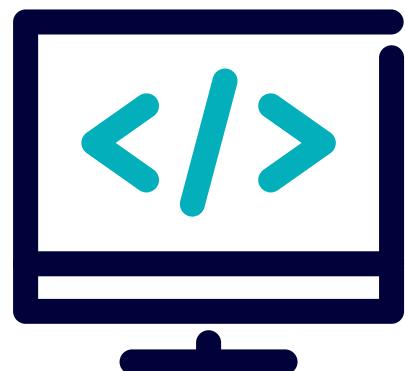


As of Apache Kafka 2.5, there is a related broker-side configuration, `fetch.max.bytes`. The effective maximum size of any fetch request will be the minimum of `fetch.min.bytes` and this value. The default value for `fetch.max.bytes` is 55 megabytes. Fetch requests from replicas will also be affected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

# Lab: Exploring Producer Performance

Please work on **Lab 8a: Exploring Producer Performance**

Refer to the Exercise Guide



hitesh@datacouch.io

# 8b: How Do Produce and Fetch Requests Get Processed on a Broker?

## Description

Thread pools. Queues. Purgatory. The path of a produce request from broker receipt to acknowledgement. The path of a fetch request from broker receipt to response with data.

hitesh@datacouch.io

# Review

We know:

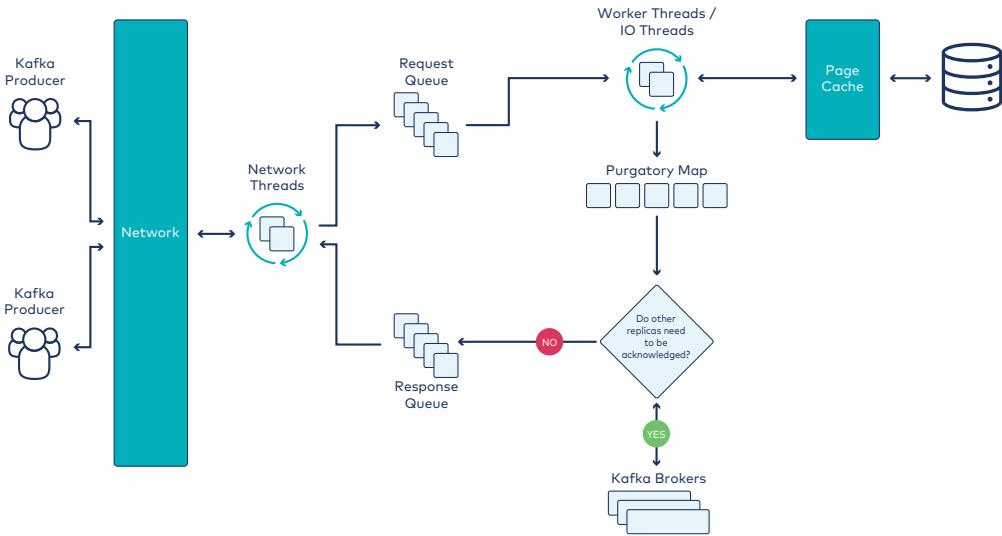
- Producers prepare messages to send.
- Producer settings like `linger.ms` and `batch.size` control how messages get grouped in batches to send.
- Batches of messages may be compressed, according to producer setting `compression.type`.
- We can configure producer property `acks` to have producers request to hear back from Kafka when messages are successfully written.

When a producer sends a batch, we say it is creating a **produce request**. So...

- What does a broker do when it receives a produce request?
- How exactly does a broker satisfy a producer's `acks` request?

Let's find out...

# Anatomy of a Produce Request



Brokers have **network threads** to receive incoming requests.



- The number of threads is configurable in broker setting `num.network.threads`.
- Default: 3.

The broker may not be able to process requests the moment they are received.

Network threads write requests into a **request queue**.



- The request queue size is `queued.max.requests`.
- Default: 500.

Another thread pool, **worker threads** or **IO threads**, contains threads to process the request queue and write messages to the page cache.



Configurable in `num.io.threads`. Default 8.

There is also a **response queue**.



Network threads handle the response queue and the request queue.  
Notice some arrows are two-way.

Finally, there is a structure called purgatory for holding produce requests that are not yet complete

hitesh@datacouch.io

# Handling Replication and **acks = all**: Purgatory

## Purgatory

Structure in memory for holding produce requests that are not yet complete

---

Purgatory 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.

hitesh@datacouch.io

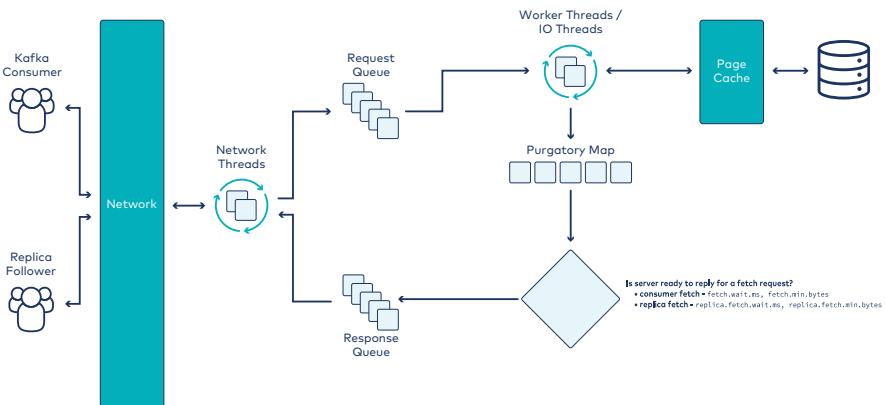
## Questions & Segue

We've seen how brokers handle produce requests. Now, what about fetch requests?

**Question:** What Kafka entities would make a fetch request?

hitesh@datacouch.io

# Anatomy of a Fetch Request



The queues and thread pools here are exactly the same as we saw before.



The produce purgatory and fetch purgatory are separate.

The anatomy is similar.

Note that the purgatory for fetch requests is a separate structure from that for produce requests. The fetch purgatory is for requests waiting for more data to satisfy the fetch request parameters.

## Activity: Looking for Broker Performance Concerns



### Discuss:

1. You might hear someone say or read that one should pay attention to garbage collection on Kafka brokers. Why do you think this is a concern?
  2. Study the anatomy of a produce request figure on the previous page. Where do you see there being potential problems/bottlenecks?
- 

hitesh@datacouch.io

# 8c: How Can You Measure and Control How Requests Make It Through a Broker?

## Description

Metrics and tuning settings for thread pools and queues in the broker's request anatomy.  
Measuring request latency overall and at stages.

hitesh@datacouch.io

# 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
Followers 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:

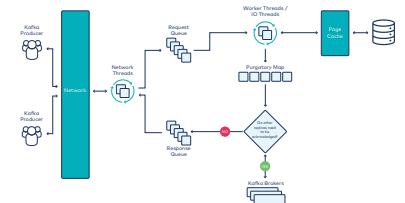
- **ProduceRequests**: write requests from producers
- **FetchRequest**s: read requests from consumers
- **ReplicaFetchRequest**s: replication requests from brokers hosting followers

Brokers process these requests in parallel using threads.

# Performance Tuning the Thread Pools



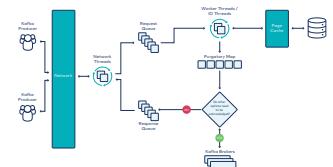
- Each thread pool is configurable
  - `num.network.threads` (default: 3, increase for SSL)
  - `num.io.threads` (default: 8)



# Performance Tuning the Request Queue



- The size of the request queue is `queued.max.requests`
  - Default is 500
  - Consider the number of clients and brokers
- 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, the 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.

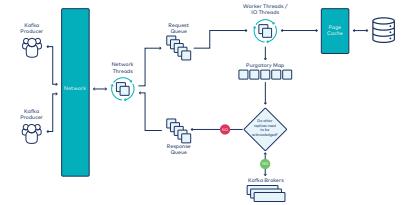
The `num.io.threads` default value is 8.

# 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.



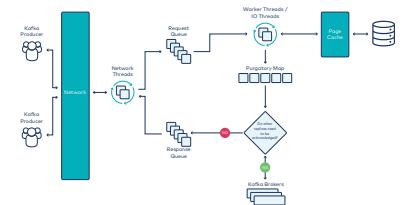
The metrics occasionally show values >1.0 due to rounding errors.

# 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
```

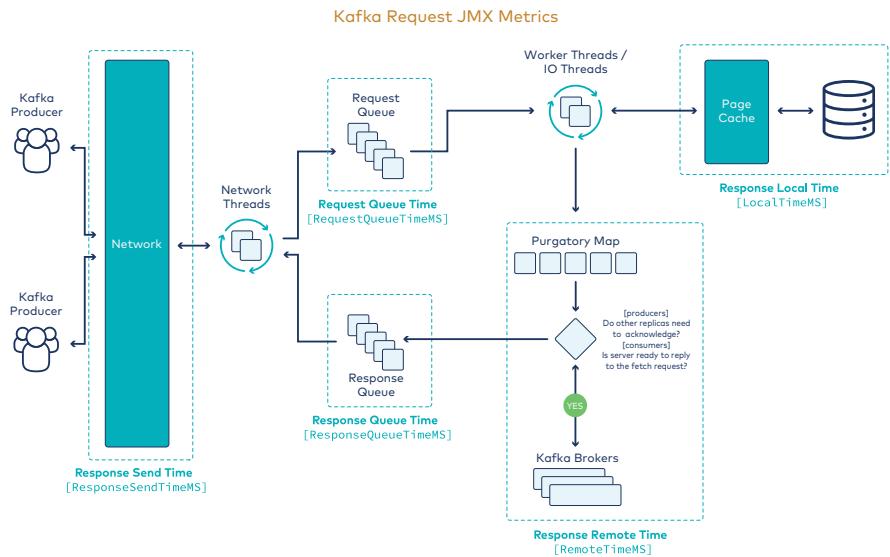
---

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 slowdown at any of them will increase the overall latency for the request.

# Request Lifecycle and Latencies

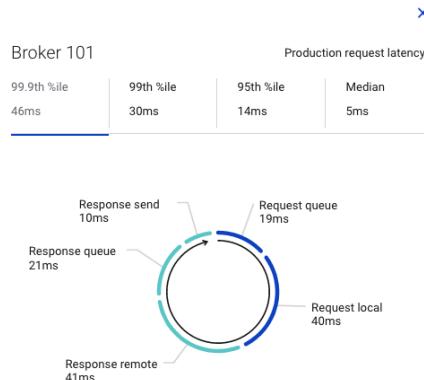
Break down **TotalTimeMs** further to see the entire request lifecycle:

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

hitesh@datacouch.io

# Request Lifecycle and Latencies

Confluent Control Center:



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

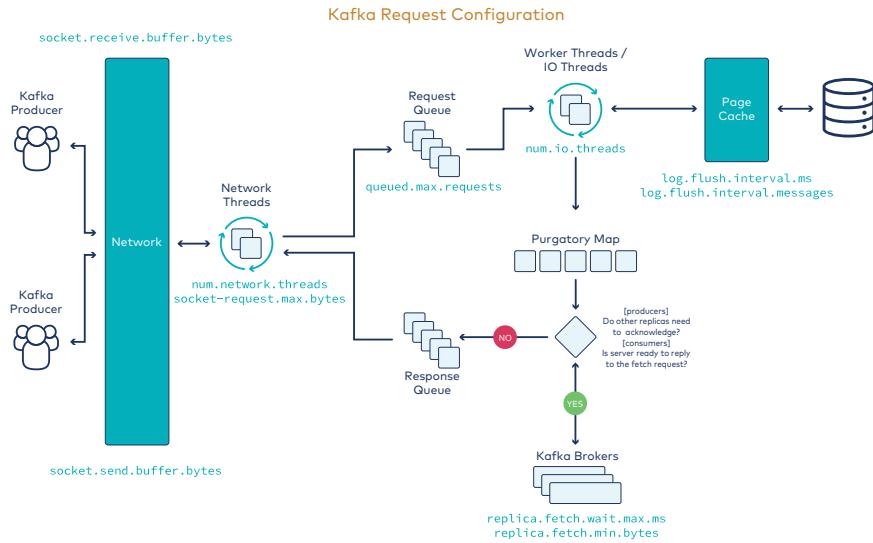
Once you have determined that the total time for the request is too large, isolate the bottleneck by viewing the metrics for each of the components. Since downstream backups can affect components earlier in the sequence, consider checking the metrics in the order:

- `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.

# 8d: What Else Can Affect Broker Performance?

## Description

Other monitoring on the broker. Message size. Garbage collection considerations. Troubleshooting exercise.

hitesh@datacouch.io

# 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

---

**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.

# Message Size Limit

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

Brokers or Topics	Replication on Brokers	Consumers
<ul style="list-style-type: none"><li>• <code>message.max.bytes</code> (B)</li><li>• <code>max.message.bytes</code> (T)<ul style="list-style-type: none"><li>◦ maximum size of message that the broker can receive from a producer (Default: 1 MB)</li></ul></li></ul>	<ul style="list-style-type: none"><li>• <code>replica.fetch.max.bytes</code><ul style="list-style-type: none"><li>◦ maximum amount of data per-partition that brokers send for replication (Default: 1 MB)</li></ul></li></ul>	<ul style="list-style-type: none"><li>• <code>max.partition.fetch.bytes</code><ul style="list-style-type: none"><li>◦ maximum amount of data per-partition the broker will return (Default: 1 MB)</li></ul></li></ul>

The configurations shown are the settings related to message size. The "B" indicates the broker setting and the "T" indicates the topic override. Kafka is optimized for the default 1 MB 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).

The settings `max.partition.fetch.bytes` (maximum bytes returned per broker, per partition) and `replica.fetch.max.bytes` (similar but for replication) are 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.

# Avoiding Soft Failures from Garbage Collection

- Garbage Collection (GC) 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 server timeouts with JMX:

```
kafka.controller:type=KafkaController,name=TimedOutBrokerHeartbeatCount
```

---

Garbage Collection (GC) 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 Controller property `broker.session.timeout.ms` (9 seconds by default), the broker will appear to be offline and all partitions it was hosting a leader will go through elections.

Also, if a consumer has a session timeout with the broker running its consumer group coordinator, it will leave and rejoin its consumer group, potentially triggering a consumer group rebalance.

The broker setting `broker.session.timeout.ms` determines how long a broker can be offline before it is considered dead (default 9 seconds). Monitor `TimedOutBrokerHeartbeatCount` on the Active Controller for any value increase.

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 for troubleshooting. If brokers are timing out from the Controller (shown as session expiration errors in the `server.log`), there is likely a 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 (the other common reason is network issues).
- Parse and analyze the GC logs.
- Check that the connection between brokers and Controller Quorum Voters is good. Otherwise, the Active Controller may falsely detect a broker as dead.

For more information about tuning garbage collection, see [this documentation](#).

hitesh@datacouch.io

# Recovering from a Broker Failure

If cluster lives on:

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

When 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 whose leader was on the broker 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. The time to complete all the elections is proportional to the number of partitions per broker. In the case that the failed node was in mixed mode (controller+broker roles), and it was the Active Controller, a new Active Controller (or Controller Leader) is elected using the KRaft algorithm, which is based on sending vote messages between members of the quorum, so the time is proportional to network latencies (note that in the legacy Zookeeper mode, the newly appointed Controller would have to download the complete metadata from Zookeeper, so recovery time would be proportional to the number of partitions).

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.

hitesh@datacouch.io

# Activity: Assessing Production Problems



## Discussion:

### Scenario:

- 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.
- `enable.idempotence` is set to `false`.

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:

- [configuration reference](#)
- [monitoring reference](#)

# 8e: How Do You Control It So One Client Does Not Dominate the Broker Resources?

## Description

Client quotas: motivation, configuration, monitoring.

hitesh@datacouch.io

# Ensure High Performance with Quotas

- High volume clients can result in:
  - Monopolizing broker resources
  - Network saturation
  - Denial of Service (DoS) to 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.

hitesh@datacouch.io

# 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.

hitesh@datacouch.io

## 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 1 KBps

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

---

The best practice is to 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.

## 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

hitesh@datacouch.io

# 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.

hitesh@datacouch.io

# 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

hitesh@datacouch.io

# Activity: Assessing Consumption Problems



## 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?

hitesh@datacouch.io

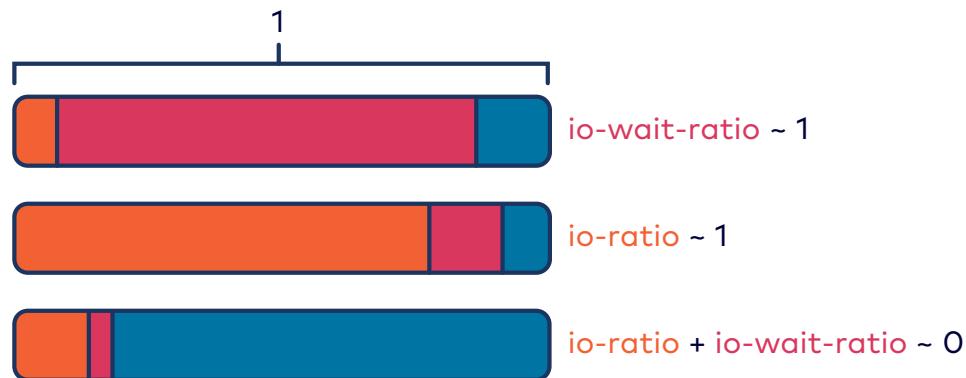
## 8f: What Should You Consider in Assessing Client Performance?

### Description

IO ratio and IO wait ratio. Implications. Other JMX metrics for clients.

hitesh@datacouch.io

# Monitoring Client Performance



- **io-ratio** The fraction of time that the client does network I/O
- **io-wait-ratio** The fraction of time that the client is idle
- **(the rest)** The fraction of time the client processes data

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.

hitesh@datacouch.io

# 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 [this documentation](#).

# Monitoring Consumer Performance

- Proactively monitor for slow consumers
- `kafka.consumer:type=consumer-fetch-manager-metrics,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.

## Activity: Understanding Client Metrics



Review the **Monitoring Client Performance** information a few pages back. Consider these scenarios:

1. Suppose `io-wait-ratio` is 0.85. Does this indicate everything is happy? If not and you think this indicates a problem, would you investigate brokers, consumers, or producers?
2. Suppose you have a consumer that takes a very long time to process records that it receives from fetches and everything else is working smoothly. In the pictures, which color would you expect to dominate?
3. A producer has batching turned off.
  - a. Which producer setting/value would cause this?
  - b. In the pictures on the referenced page, which color would dominate? Also express this in terms of a number for one of the named metrics.

## 8g: How Can You Test How Clients Perform?

### Description

CLI client performance testing tools and their use.

hitesh@datacouch.io

# 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.

# How to Test Performance

Determine `p`: producer throughput per partition

- Run a single producer on a single server
- `kafka-producer-perf-test`

Determine `c`: consumer throughput per partition

- Run a single consumer on a single server
- `kafka-consumer-perf-test`

---

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.

# 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

---

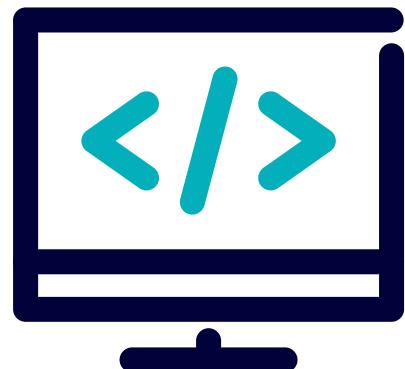
Kafka provides inbound and outbound metrics for bytes and messages on a broker. These data points are also available on a per-topic basis.

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

# Lab: Performance Tuning

Please work on **Lab 8b: Performance Tuning**

Refer to the Exercise Guide



hitesh@datacouch.io

# 9: Securing a Kafka Cluster



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 4 lessons:

- What are the Basic Ideas You Should Know about Kafka Security?
- What Options Do You Have For Securing a Kafka/Confluent Deployment?
- How Can You Easily Control Who Can Access What?
- What Should You Know Securing a Deployment Beyond Kafka Itself?

Where this fits in:

- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite: Managing a Kafka Cluster

# Learning Objectives



Completing this lesson and associated exercises will enable you to:

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

hitesh@datacouch.io

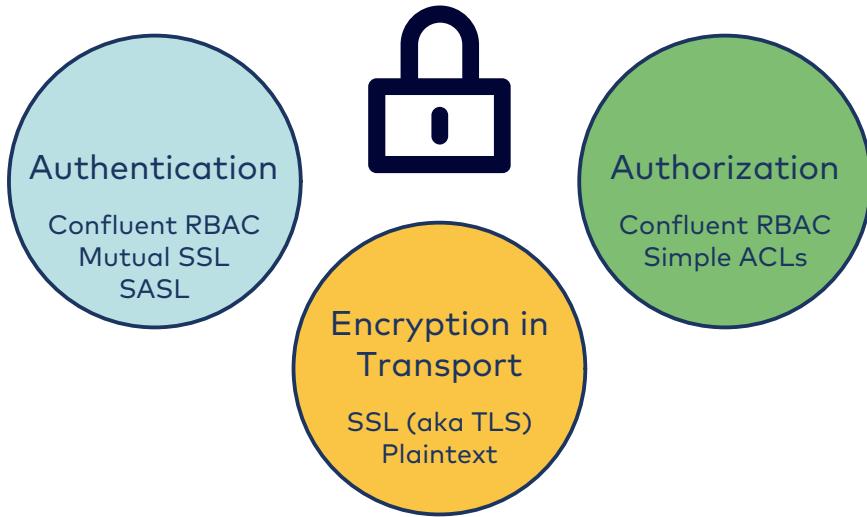
# 9a: What are the Basic Ideas You Should Know about Kafka Security?

## Description

Overview of security in Kafka. Authentication vs. authorization. Encryption. Points of vulnerability.

hitesh@datacouch.io

# 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, you 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.

---

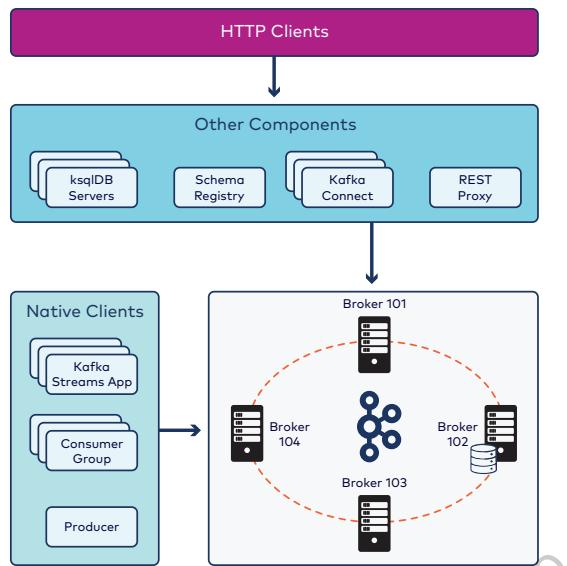
- Transport
  - Data can be transported without encryption as **PLAINTEXT**, or with encryption using SSL (Secure Socket Layer).
- Authentication
  - By default, the SSL protocol authenticates a server to a client (one-way authentication), but it also allows the client to authenticate to the server as well, which is called mutual SSL and, for superseding protocol TLS, mutual TLS or simply mTLS.
  - 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.

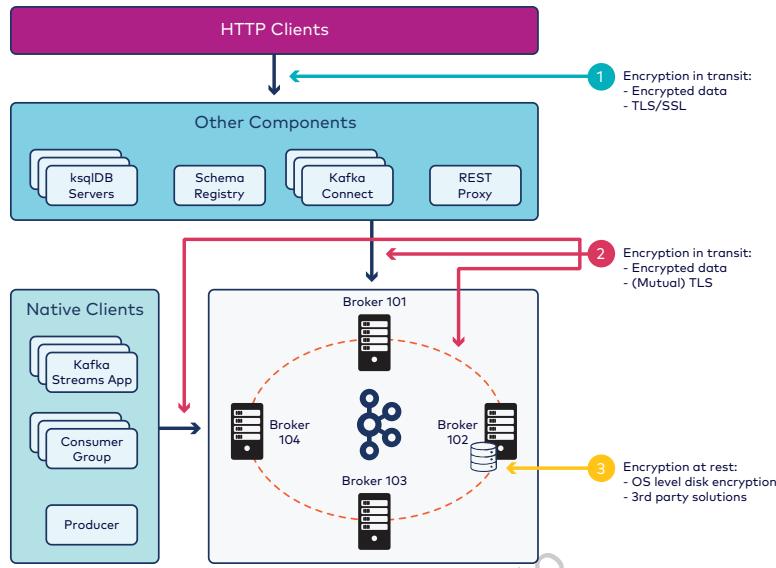
# Security - Architecture



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 one to many brokers
- On the left hand side we have the so-called native Kafka clients: producers, consumer groups and Kafka Streams applications (they only use the Kafka Protocol)
- On top, we have other components that expose some HTTP/REST protocol on one side and use the Kafka Protocol on another: Kafka Connect, Confluent Schema Registry, Confluent REST Proxy and Confluent ksqlDB applications.
- At the very top, in purple we have the HTTP clients that use the previous REST APIs

# Security - Encryption at Rest & in Transit



The Kafka cluster can be secured as follows:

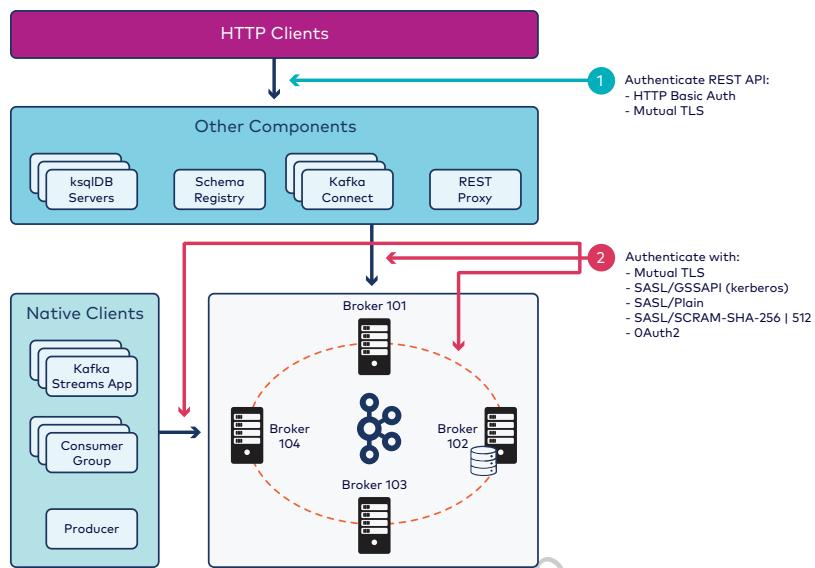
1. Encryption between HTTP clients and Confluent Platform services such as Schema Registry happens in the following ways
  - Using TLS/SSL
  - Using End-To-End Encryption (not managed by Kafka)
2. **In-Flight** Encryption (or **in-transit**) is mainly achieved in the following 2 ways:
  - End-to-End Encryption: the Producer encrypts the data before sending it to the Broker. The Consumer decrypts this data (KMS etc. not managed by Kafka)
  - using (mutual) TLS. This applies for both situations:
    - client → broker
    - broker → broker
3. Encryption at rest is implemented either by using OS level disk partition encryption or by using 3rd party services



End-to-End encryption involves correct management of key distribution, and extra code in all the clients involved. Regarding ksqlDB, an engineer from Confluent has shown in a proof-of-concept that it can be done, even at the field level, by creating some UDFs that integrate such extra code.

hitesh@datacouch.io

# 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 in-flight (HTTPS). This module focuses on security with Kafka clients and brokers rather than REST API clients, so we only briefly mention REST API security here.



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](#)
- [Confluent Control Center HTTPS](#)
- [Kafka Connect HTTPS](#) \*<https://docs.confluent.io/current/schema-registry/security.html#additional-configurations-for-https> [Schema Registry HTTPS]
- [REST Proxy HTTPS](#)
- [KSQL HTTPS](#)

Kafka clients (including the REST API **servers** just mentioned) can authenticate with the Kafka cluster with SASL or Mutual SSL.

# 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



Brokers may need to set up **advertised.listeners** in addition to **listeners** when the hostname/IP/port resolved by clients is different from those resolved by brokers themselves (e.g. NAT, aliasing, ...)

As we think about the various means of authentication and authorization we should understand the ports used by the various services.

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 fail over 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](#).

# An Advanced Listeners Config Example

We can 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
```

hitesh@datacouch.io

# 9b: What Options Do You Have For Securing a Kafka/Confluent Deployment?

## Description

Survey of security options.

hitesh@datacouch.io

## SSL/TLS or SASL Manual Configuration

- Free
- But you have to do all the work

- You will experience this in lab.



See the appendix for examples and details regarding SSL/TLS and SASL.

hitesh@datacouch.io

# Confluent RBAC

- **Role Based Access Control**
  - Paid Confluent feature
  - Includes both authentication and authorization...
  - ...by defined roles
- 

RBAC is a relatively simple concept that has been around for some time. A user is bound to a Role and the Role has a set of privileges. We do **not** cover RBAC in this course, but you have it documented [here](#)

hitesh@datacouch.io

# Security - Confluent Cloud

- Security enabled **out-of-the-box**
  - 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

# 9c: How Can You Easily Control Who Can Access What?

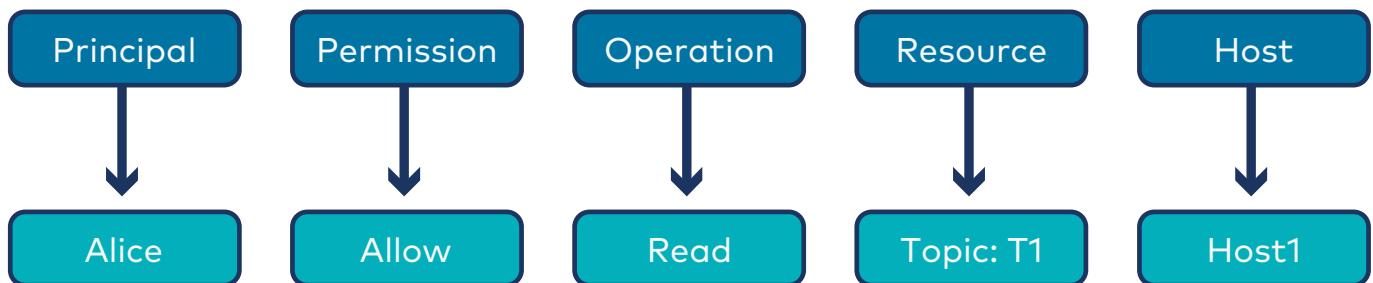
## Description

Components of Kafka ACL entries. How to add and remove ACLs. Wildcards.

hitesh@datacouch.io

# Access Control Lists (ACLs)

ACL 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.

# 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 the brokers.

Both the type name (**User**) and the principal names are case-sensitive. The principal shows above wouldn't match user **alice**.

hitesh@datacouch.io

# 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.

hitesh@datacouch.io

# 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>	<code>Topic</code>
<code>Read</code>	<code>ConsumerGroup</code>
<code>Create, ClusterAction</code> intra-cluster operations (leader election, replication, etc.)	<code>Cluster</code>

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.

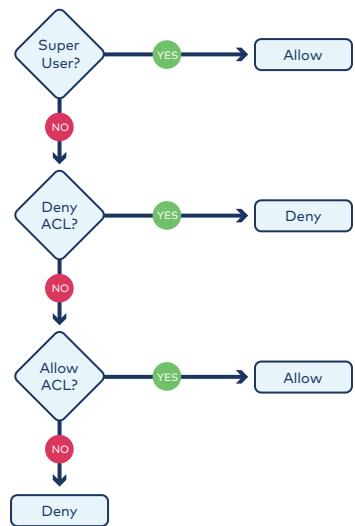
The complete list of operations is here: [https://docs.confluent.io/platform/current/kafka\\_authorization.html#operations](https://docs.confluent.io/platform/current/kafka_authorization.html#operations)

## Hosts

- Allows firewall-type security, even in a non-secure environment
  - Without needing system/network administrators to get involved

hitesh@datacouch.io

# Permission Check Sequence



hitesh@datacouch.io

# 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 the cluster metadata topic and propagated to brokers asynchronously
- ACLs are cached in the broker for better performance
- Make Kafka principal superusers
  - 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 [here](#).

ACLs are stored in the metadata topic and cached on each broker so that look-ups can be done locally.

Superusers are configured using the `super.users` setting in `server.properties` (in all nodes).



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
```

hitesh@datacouch.io

# 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.

# 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
```

hitesh@datacouch.io

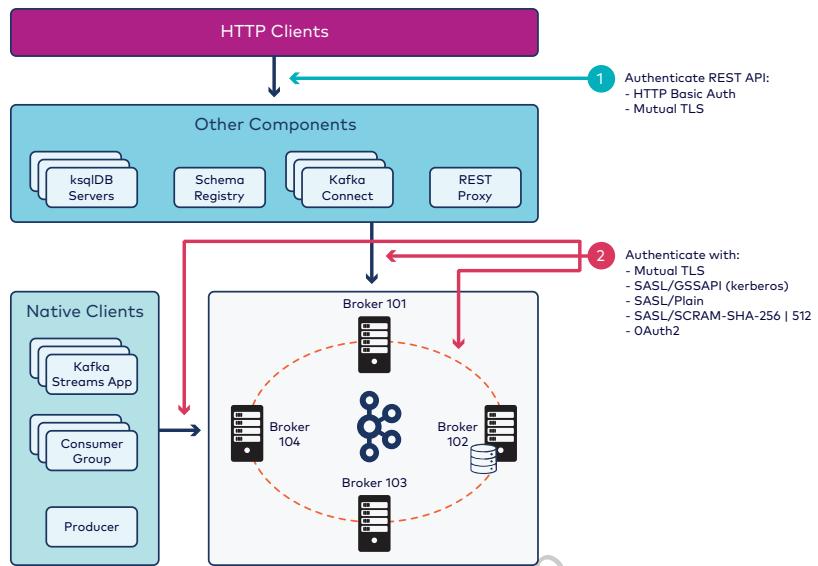
# 9d: What Should You Know Securing a Deployment Beyond Kafka Itself?

## Description

Securing the whole environment.

hitesh@datacouch.io

# 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](#)
- [Schema Registry documentation](#)
- [Schema registry security plugin](#)
- [Schema Registry configuration reference](#)

## 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 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](#)
- [REST Proxy HTTPS](#)
- [REST Proxy security plugin](#)
- [REST Proxy configuration reference](#)

## Version notes:

- Since Confluent Platform 4.0: REST proxy security plugin propagates client principal authentication to Kafka brokers

# 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

---

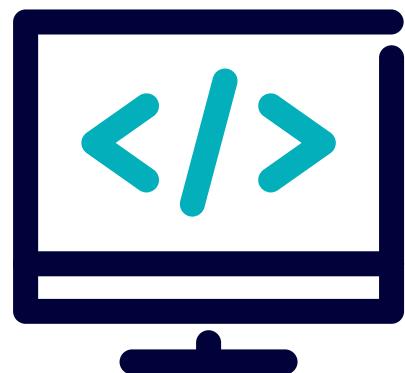
This procedure assumes that the cluster's mission-critical topics are replicated so that a rolling reboot can be performed to implement the changes to the **server.properties** files.

hitesh@datacouch.io

# Lab: Securing the Kafka Cluster

Please work on **Lab 9a: Securing the Kafka Cluster**

Refer to the Exercise Guide



hitesh@datacouch.io

# 10: Understanding Kafka Connect



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 4 lessons:

- What Can You Do with Kafka Connect?
- How Do You Configure Workers and Connectors?
- Deep Dive into a Connector & Finding Connectors
- What Else Can One Do With Connect?

Where this fits in:

- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite: Consumer Groups and Load Balancing

# Learning Objectives



Completing this lesson and associated exercises will enable you 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)

hitesh@datacouch.io

# 10a: What Can You Do with Kafka Connect?

## Description

Motivating what Connect can do and why to use it over self-made solutions. Motivating how it can “factor out” common behavior yet leverage Connectors. Connectors vs. tasks vs. workers. Relating Connect to other components of Kafka and how it works at a high level, e.g., scalability, converters, offsets.

hitesh@datacouch.io

# Wanted: Data From Another System in Kafka; Kafka Data To Another System

Suppose you have

- Some data in some other system and you want to get it into Kafka
- Some data in Kafka and want to export it to another system

Your development team could program custom producers or consumers with hooks into the other system to make this happen...

But... there's a better way...

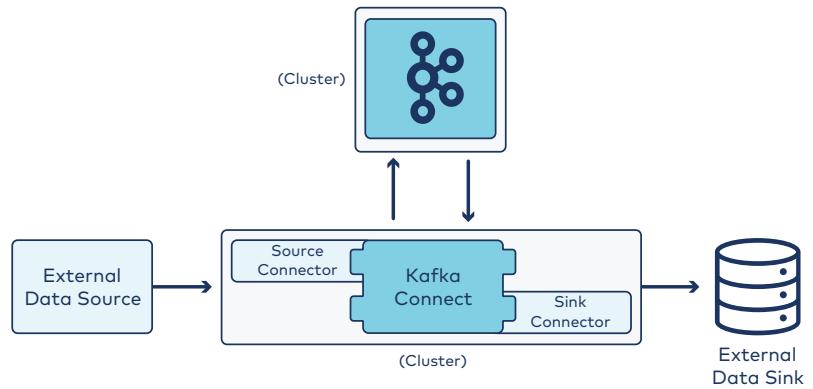
hitesh@datacouch.io

# Kafka Connect to the Rescue!

Kafka Connect does the work for us!

All copying behavior is in Kafka Connect.

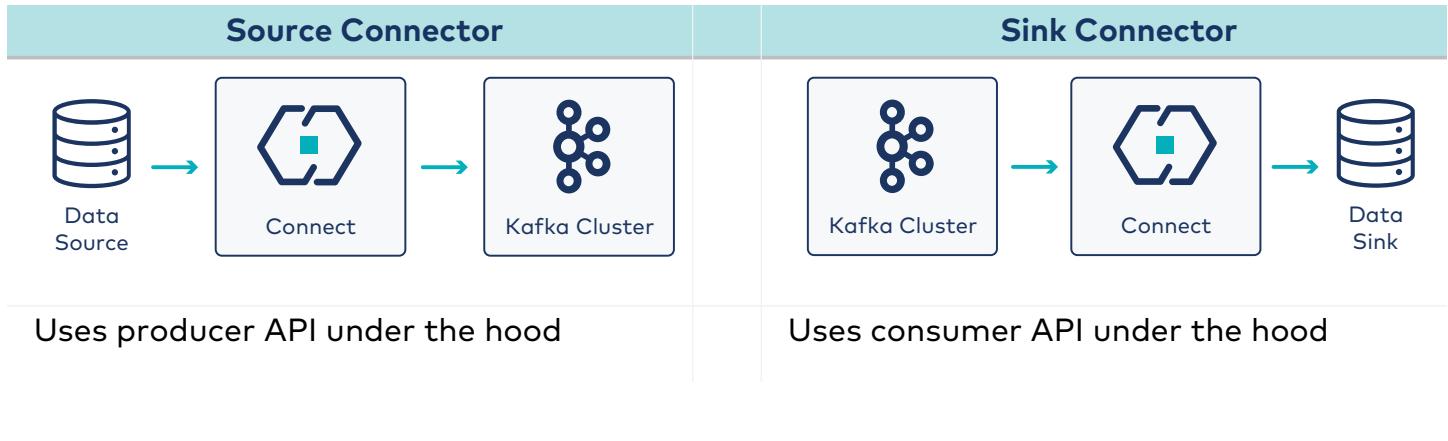
Plugins called **Connectors** contain the logic specific to particular external systems.



The Kafka Connect API is part of core Kafka.

# Sources and Sinks

Two kinds of connectors...



We note the two kinds of Connectors here.

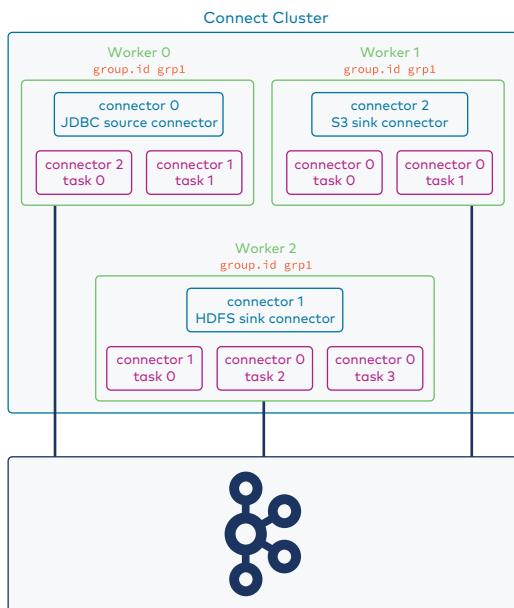
In fact, Kafka Connect is built on top of what we already know - producers and consumers.

# Players in the Kafka Connect World

<b>Kafka Connect</b>	<ul style="list-style-type: none"><li>• A connector has<ul style="list-style-type: none"><li>◦ one or more tasks</li></ul></li></ul>
<b>Connector</b>	<ul style="list-style-type: none"><li>• A worker runs<ul style="list-style-type: none"><li>◦ zero or more connectors</li><li>◦ zero or more tasks</li></ul></li></ul>
<b>Task</b>	<ul style="list-style-type: none"><li>• A connector has<ul style="list-style-type: none"><li>◦ one or more tasks</li></ul></li></ul>
<b>Worker</b>	<ul style="list-style-type: none"><li>• A worker runs<ul style="list-style-type: none"><li>◦ zero or more connectors</li><li>◦ zero or more tasks</li></ul></li></ul>
	<p>unit of parallelism into which connector copying logic is broken up</p>
	<p>process that runs connectors and/or tasks</p>

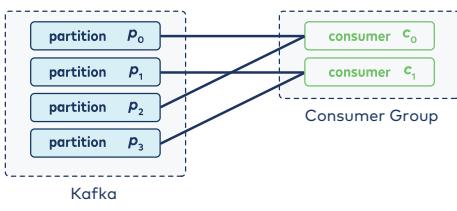
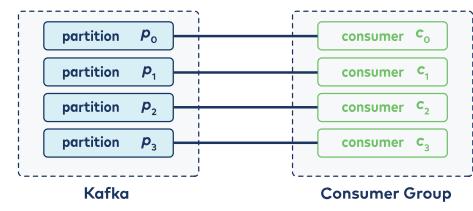
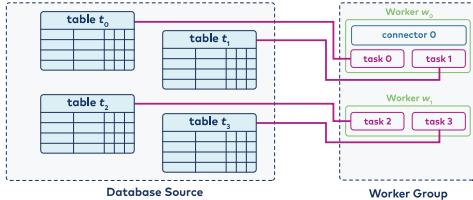
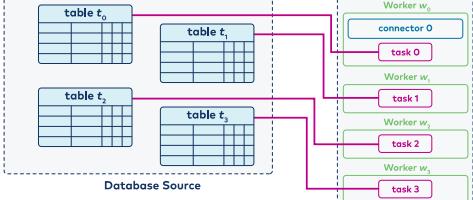
We'll use these terms throughout the module. An objective of this lesson is to learn them and the relationships between them. The activity at the end of the lesson will reinforce this.

# Example of a Connect Cluster



Here is an example illustrating a Connect cluster. We see workers running connectors - both source and sink connectors - and tasks. Kafka's group management protocol handles which connector(s) and task(s) are running on each worker.

# Groups Again!

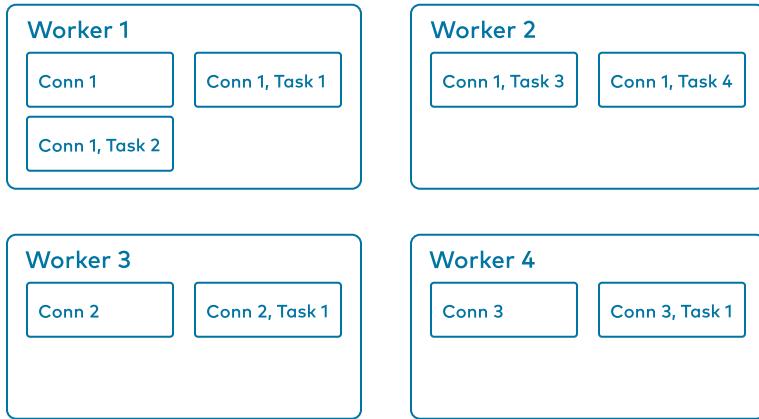
	Before Scaling	After Scaling
Consumer Group		
Worker Group		
<p><b>i</b> Like with consumers, we can add workers to groups and get automatic balancing.</p>		

Recall the module Groups, Consumers, and Partitions, especially the first two lessons. Workers live in groups just like consumers, and group management happens in the same way. There is automatic rebalancing when we scale up or a worker dies. Workers heartbeat to Kafka in the same way as consumers, governed by the same `heartbeat.interval.ms` and `session.timeout.ms` settings. (Defaults are 3 and 45 seconds, respectively, for consumers, but there is the inconsistency that the default `session.timeout.ms` for workers is 10 seconds).

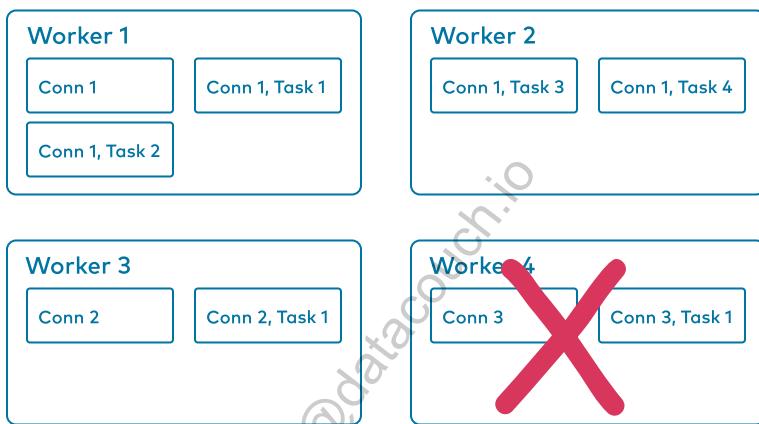
Technically, the workers are running tasks, which in turn are connected to the tables.

We'll go into configuring how many tasks in the next section.

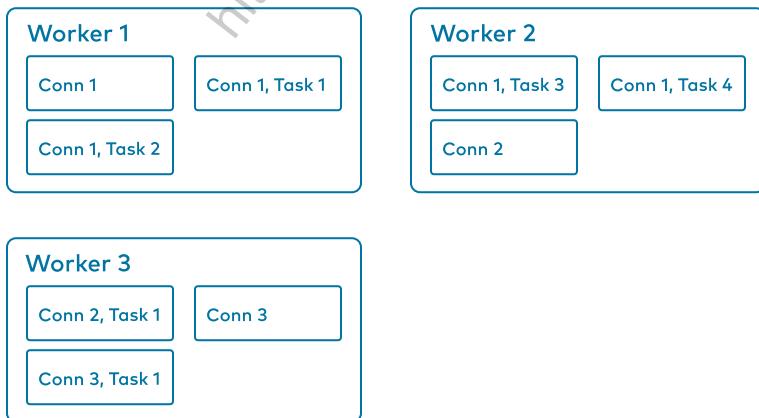
**Example:** Suppose we start with a situation like this:



Then a worker fails:



Automatic rebalancing might yield this:



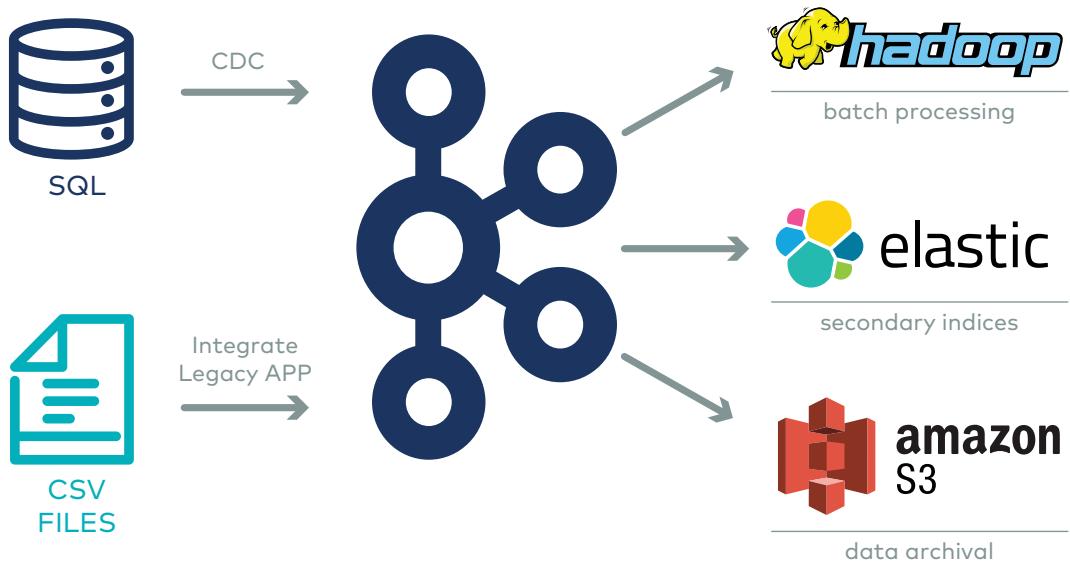
# Powered by Kafka, and Behaving Like Kafka

- Same group management protocol
    - Failure detection
    - Scaling up and down
  - Producer and consumer under the hood
  - Sink connectors maintain consumer offsets → **sink offsets**
  - Source connectors track **source offsets**
  - Data must be serialized and deserialized → **converters**
- 

Here we see many of the ideas we know from producers and consumers showing up in the Kafka Connect world.

hitesh@datacouch.io

# 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)

# Activity: Reviewing Kafka Connect Concepts



## Quick Matching Game

For each item on the left, identify which items on the right apply

- |              |                                   |
|--------------|-----------------------------------|
| 1. Connector | a. unit of parallelism            |
| 2. Task      | b. can be part of a group         |
| 3. Worker    | c. like a serializer              |
| 4. Converter | d. relates to one or more tasks   |
|              | e. like a deserializer            |
|              | f. specific to an external system |
|              | g. could run a connector          |



Not all connectors support multiple tasks and parallelism. For example, the **syslog** source connector only supports one task.

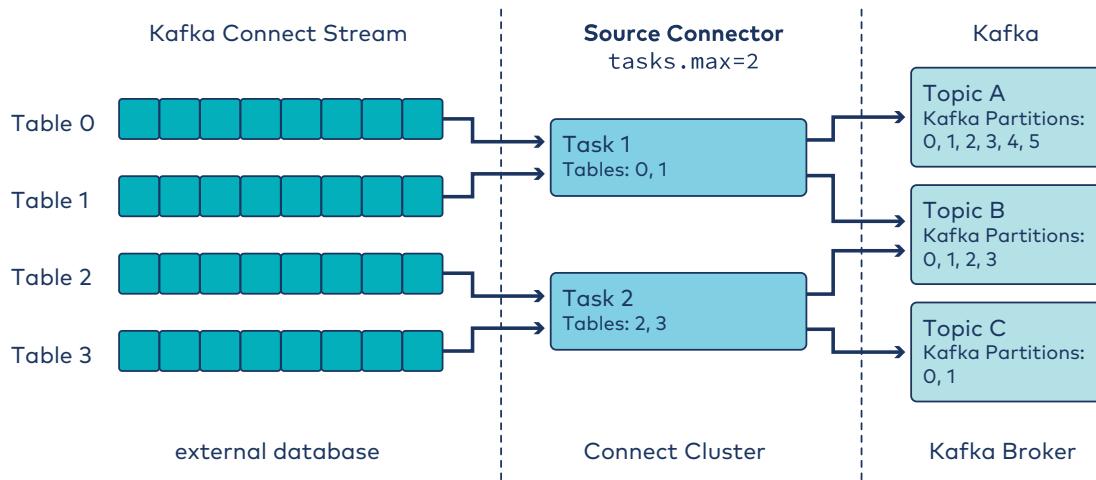
# 10b: How Do You Configure Workers and Connectors?

## Description

Configuration of workers in distributed mode and configuration of connectors in general. Quick overview of standalone mode differences.

hitesh@datacouch.io

# Providing Parallelism & Scalability



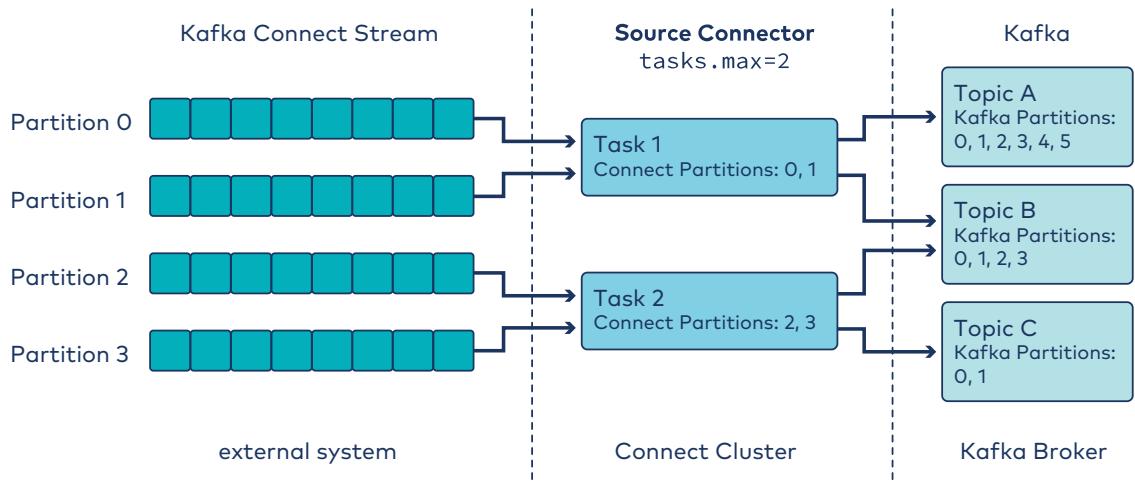
So

- 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 tables 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.

This image is in terms of a database source connector. We could generalize to "Connect Partitions" from the tables of the database.

We can generalize the above graphic:



# What Do We Need to Configure?

Remember:

- A **connector** applies to a particular external source or sink
- A **connector** may be broken into one or more parallel **tasks**
- A **worker**...
  - ... runs zero or more connectors
  - ... runs zero or more tasks
  - ... is generally part of a group, managed by Kafka's group management protocol

## Activity: Brainstorming Connector Configurations



What do you think we need to specify to configure a **connector**? Discuss with a small group for 2 minutes.

# Configuring Connectors

Name	Description	Default
<code>name</code>	Connector's unique name	
<code>connector.class</code>	Name of the Java bytecodes file for the connector	
<code>tasks.max</code>	Maximum number of tasks to create - if possible	1
<code>key.converter</code>	Converter to (de)serialize keys	(worker setting)
<code>value.converter</code>	Converter to (de)serialize values	(worker setting)
<code>topics</code>	For <b>sink connectors only</b> , comma-separated list of topics to consume from	

Note that `tasks.max` is a limit and is restricted by the shape of the data. If Kafka Connect cannot achieve the desired number of tasks, then it will create as many as possible. For example, if you have a database source connector for a database with 4 tables but set `tasks.max` to 6, you will get 4 tasks, because the copying cannot be parallelized further.

Note that while you can work with connectors in Confluent Cloud, as of August 2022, it is not possible to configure connectors directly through ksqlDB.

It is also 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>	<code>&gt;= 1</code> for a specific valid value, or <code>-1</code> to use the broker's default value
<code>topic.creation.\$alias.partitions</code>	<code>&gt;= 1</code> for a specific valid value, or <code>-1</code> to use the broker's default value
<code>topic.creation.\$alias.\${kafkaTopicSpecificConfigName}</code>	List of input topics to consume from

# What Do We Need to Configure? (2)

Remember:

- A **connector** applies to a particular external source or sink
- A **connector** may be broken into one or more parallel **tasks**
- A **worker**...
  - ... runs zero or more connectors
  - ... runs zero or more tasks
  - ... is generally part of a group, managed by Kafka's group management protocol

We've established what we need for connectors. So...

## Activity: Brainstorming Worker Configurations



What do you think we need to specify to configure a **worker**? Discuss with a small group for 2 minutes.

# Configuring a Worker

Name	Description	Default
<code>bootstrap.servers</code>	List of host:port pairs to connect	
<code>group.id</code>	Identifier of what group this worker is a member of	
<code>heartbeat.interval.ms</code>	How frequently heartbeats are sent	3 sec.
<code>session.timeout.ms</code>	Time after which a worker that does not heartbeat is deemed dead	10 sec.
<code>key.converter</code>	Converter to (de)serialize keys	
<code>value.converter</code>	Converter to (de)serialize values	
<code>topic.creation.enable</code>	Whether source connectors are permitted to create topics	<code>true</code>

For more: [this documentation](#).

We see converter configs here and for connectors. Converter configs can be set at the worker level to apply to all connectors running on a worker. They can be overridden at the connector level too.

Another configuration setting to consider is `client.id`; like with producers and consumers, this is a way of naming the client, worker in this case, so it is distinguished in monitoring tools and system logs.

## Configuring a Worker: Topics

Name	Stores	Default Num Partitions
<code>config.storage.topic</code>	Connector and task configuration	1
<code>offset.storage.topic</code>	Source and sink offsets	25
<code>status.storage.topic</code>	Current status of connectors and tasks, e.g., running, paused, etc.	5

Kafka Connect uses a few internal topics for configuration settings too. You can configure what those topics are called.

The topics are automatically configured with recommended replication factor and partition count values, and they are compacted. The number of partitions is shown in the table.

If you manually configure these topics, keep the relative partition counts in mind.

# Running the Config

- Create a properties file for Connect, e.g., `connect-distributed.properties`
- Run on **each** worker node:

```
$ connect-distributed connect-distributed.properties
```

- Can configure Connectors via REST API
  - Or, indirectly, via Confluent Control Center
  - You will see this in lab!
- Can configure connectors via ksqlDB as well

---

In your configuration file, list off properties and their values, e.g.,

```
bootstrap.servers=kafka1:9092, kafka2:9092, kafka3:9092
```

[Connect REST Interface documentation](#)

# Standalone Mode

- We've looked at Kafka Connect in **distributed mode**  
→ Wanted in production in most cases
- There is a **standalone mode** too
  - Good for development and testing
  - Needed for certain connectors
- Standalone config differences:
  - Offsets are stored in a file rather than in a Kafka topic. Filename is set in `offset.storage.file.filename`

---

Some connectors, e.g., the [Syslog Source Connector](#) require being run in standalone mode.

hitesh@datacouch.io

# 10c: Deep Dive into a Connector & Finding Connectors

## Description

Details of the JDBC Source Connector, configuration details, working through why one would do certain configs with examples. Finding Connectors on Confluent Hub.

hitesh@datacouch.io

## 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.

Note that the JDBC source connector does not generate keys by default, according to [this documentation](#). (That link provides a workaround too.)

## Query Mode (1)

Incremental query mode	Description
Bulk	Load all rows in the table. Does not detect new or updated rows.

The connector can detect new and updated rows in several ways, but let's start simple: for a one-time load, not incremental, unfiltered, we just use **bulk** mode.

hitesh@datacouch.io

## Query Mode (2)

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: Custom Query

We can also define a **custom query** to use in conjunction with the previous options for custom filtering.

---

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.

hitesh@datacouch.io

# Configuration

Property	Description
<code>connection.url</code>	The JDBC connection URL for the database
<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>topic.prefix</code>	Prefix to prepend to table names to generate the Kafka topic name
<code>table.blacklist</code>	A list of tables to ignore and not import.
<code>table.whitelist</code>	A list of tables to import.



See [JDBC Connector docs](#) for a complete list

One particular config you might also find interesting is `table.types`. You could set this to `VIEW` if you are trying to source from a view.

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 Config Example

```
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"  
14  }  
15 }
```

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 something called SMTs:

```
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"
```

This lesson is not meant to be your formal introduction to SMTs, but this is provided as an example.

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."

hitesh@datacouch.io

# Other Connectors

Search Confluent Hub at [confluent.io/hub](https://confluent.io/hub) for connectors!

The screenshot shows the Confluent Hub search interface. On the left, there are filters for 'Plugin type' (Sink, Source, Transform, Converter), 'Enterprise support' (Confluent supported, Partner supported, None), 'Verification' (Confluent built, Confluent tested, Verified gold, Verified standard, None), and 'License' (Commercial, Free). The search bar contains the query 'Kafka'. The results section displays two connectors:

- Kafka Connect GCP Pub-Sub**: A source connector for Google Cloud Pub/Sub. It is available fully-managed on Confluent Cloud. It has Confluent supported enterprise support, Confluent built verification, and a commercial license.
- Kafka Connect S3**: A sink connector for Amazon S3. It is available fully-managed on Confluent Cloud. It has Confluent supported enterprise support, Confluent built verification, and a free license.

You can find many more connectors, along with their documentation, on [Confluent Hub](https://confluent.io/hub).

# 10d: What Else Can One Do With Connect?

## Description

Case studies of using a Connector to read in data from a source, transform the data, and write it to a sink. SMTs vs. Kafka Streams vs. ksqlDB as options for transforming data.

hitesh@datacouch.io

# Development Power

Kafka Connect can be leveraged along with other tools to solve various development problems.

Common classes of problems include:

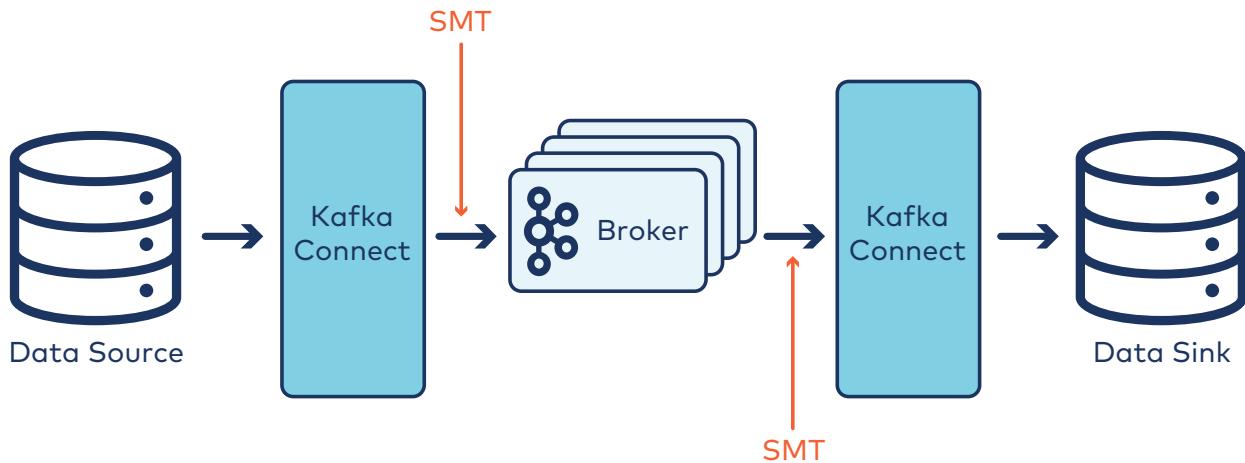
- **ETL** = Extract, Transform, Load
- **CDC** = Change Data Capture

In both cases, developers can leverage Kafka Connect to get data into Kafka.

The "transform" may be achieved either via:

- Streaming applications
  - Kafka Streams
  - ksqlDB
- Single Message Transforms...

# Single Message Transforms



Here are several SMTs:

Transform	Description
<code>InsertField</code>	insert a field using attributes from message metadata or from a configured static value
<code>ReplaceField</code>	rename fields, or apply a blacklist or whitelist to filter
<code>MaskField</code>	replace field with valid <code>null</code> value for the type (0, empty string, etc)
<code>ValueToKey</code>	replace the key with a new key formed from a subset of fields in the value payload
<code>HoistField</code>	wrap the entire event as a single field inside a <code>Struct</code> or a <code>Map</code>
<code>ExtractField</code>	extract a specific field from <code>Struct</code> and <code>Map</code> and include only this field in results
<code>SetSchemaMetadata</code>	modify the schema name or version
<code>TimestampRouter</code>	modify the topic of a record based on the original topic name and timestamp

Transform	Description
<code>RegexRouter</code>	update a record topic using the configured regular expression and replacement string

For more information on SMTs, see

<https://docs.confluent.io/current/connect/transforms/index.html>

See your student handbook in the JDBC source connector lesson for an extension of the JDBC source connector example with SMTs added.

In the Developer class, we consider this problem to motivate SMTs as a simple alternative to streaming applications:

Say we

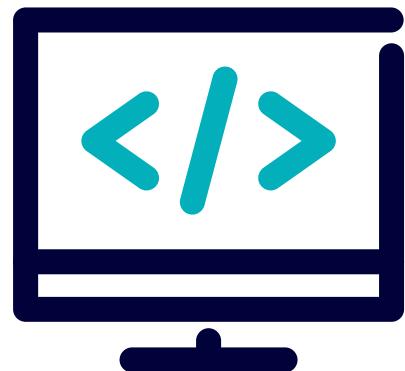
- Have access to a hospital's database.
- Want to extract information on patients who've been diagnosed with cancer.
- Want to load this information to CSV files that will be given to medical researchers studying correlations between diagnoses and patient traits.

In this case, an SMT can be used to mask PII in transforming the data.

# Lab: Running Kafka Connect

Please work on **Lab 10a: Running Kafka Connect**

Refer to the Exercise Guide



hitesh@datacouch.io

# 11: Deploying Kafka in Production



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Module Overview



This module contains 6 lessons:

Each lesson enables you to answer what Confluent advises for deploying each of the following in production:

- Kafka
- Kafka Connect
- Confluent Schema Registry
- Confluent REST Proxy
- Kafka Streams and ksqlDB
- Confluent Control Center

Where this fits in:

- Hard Prerequisite: All modules of this Administration course

# Learning Objectives

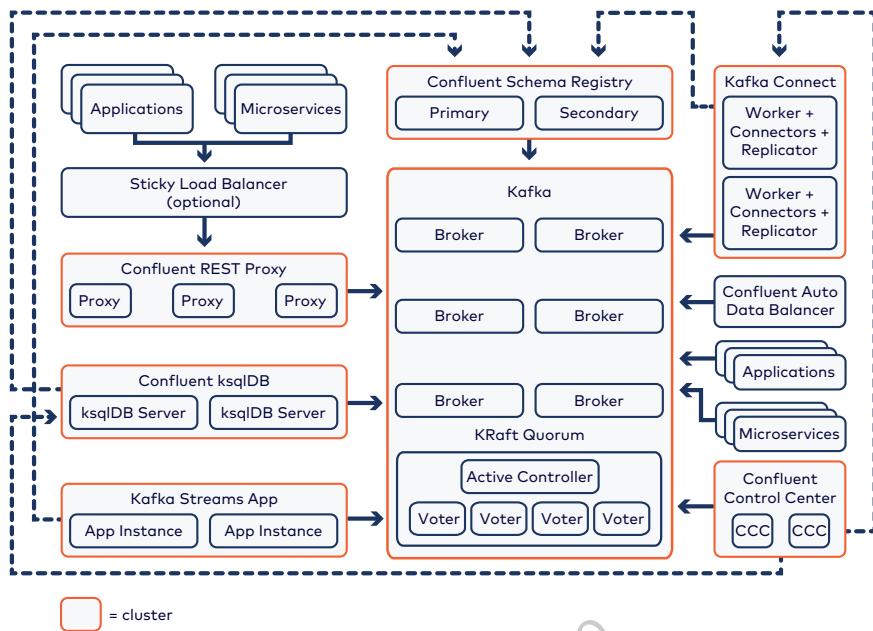


Completing this lesson and associated exercises will enable you to:

- Deploy highly available services:
  - Kafka
  - Kafka Connect
  - Confluent Schema Registry
  - Confluent REST Proxy
  - Kafka Streams and ksqlDB
  - Confluent Control Center
- Do capacity planning

hitesh@datacouch.io

# 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.

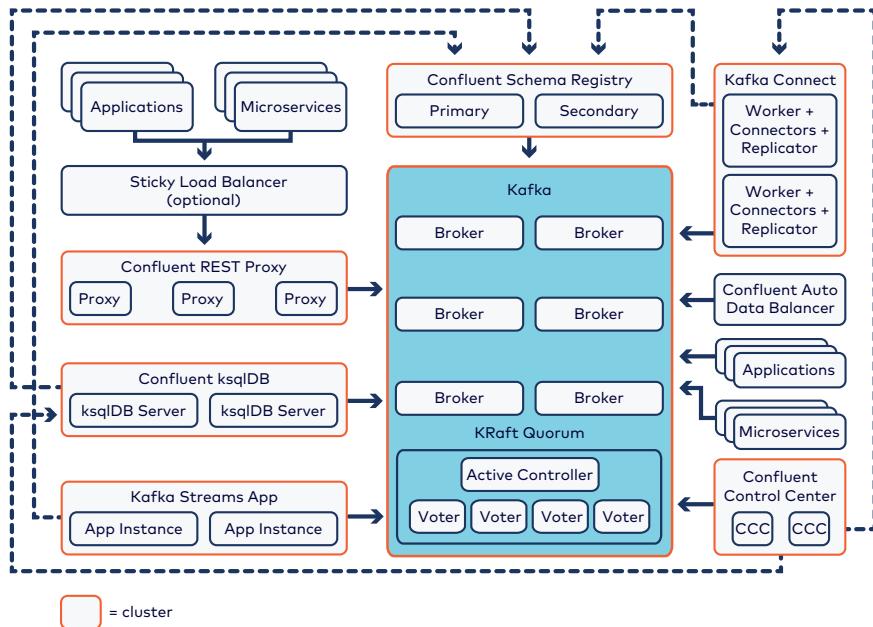
# 11a: What Does Confluent Advise for Deploying Brokers in Production?

## Description

Best practices and capacity planning for brokers in production.

hitesh@datacouch.io

# Kafka Reference Architecture: Brokers



# Broker Design

- Run on dedicated servers
- 5 controllers
  - In multi-DC / multi-AZ deployment, distribute controllers
- $n$  brokers → 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:
  1. What replication factor is acceptable for mission-critical data?
  2. How many brokers do you need to accommodate this?
  3. 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 broker and controller nodes (dedicated KRaft). Deploy 5 KRaft controller nodes (3 is for non-production clusters). Also, if deploying a multi-AZ / multi-datacenter cluster, distribute controllers too. Obviously brokers should be distributed too, but start with the distribution of controllers first.

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 `listeners / advertised.listeners`. 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. Breaking them down:

1. 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.
2. Assuming a replication factor of 3 for mission-critical topics, there would need to be at least 3 brokers.

3. Two of these 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 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.

# 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.

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**
- 

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 related to the discussion questions:

- 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.

hitesh@datacouch.io

## Network Bandwidth

- Gigabit Ethernet sufficient for smaller applications
  - 10Gb Ethernet needed for large installations
  - Enable compression on producers
  - Optional: Isolate Inter-Broker 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.

Isolating Inter-Broker communication was already common, but with the new KRaft model it also inherits advantages of the Broker-Zookeeper network isolation. The benefits are twofold:

- Less contention on the network between Broker-Client and Broker-Broker traffic
- Isolation of KRaft controllers from clients (and users) for security (e.g. DOS-ing the controllers)

## 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 1 GB
  - `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's 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.

hitesh@datacouch.io

# 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.

## Considerations:

- 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.

# 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)
```



Recommended limits: 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)

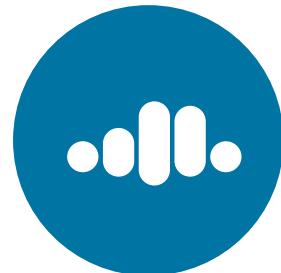
While tests have shown much better scalability of KRaft-based Kafka cluster over legacy Zookeeper-based ones, we currently don't have updated values for the soft limits shown above. To be on the safe side, it may be advisable to still keep these limits. For reference, below is the excerpt that was used for Zookeeper-based-clusters:

Typically, very large brokers can sustain a maximum of 2000-4000 partitions; most installations will be in the 1000s 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.

hitesh@datacouch.io

# Deploying Kafka in the Cloud

- Self-managed cloud deployment:
  - Memory optimized compute instances
  - Multiple availability zones (`broker.rack`)
  - Private subnet for inter-broker traffic
  - Lockdown firewall rules, Kafka security
  - For AWS: "EBS optimized" instances
- Or consider:



Virtual cores are weaker than physical cores

Leverage Kafka's rack awareness feature by assigning `broker.rack` value according to availability zone. More specific guidelines about placement of brokers are discussed in the appendix.

Distributed storage is discouraged for Kafka on-premises 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.

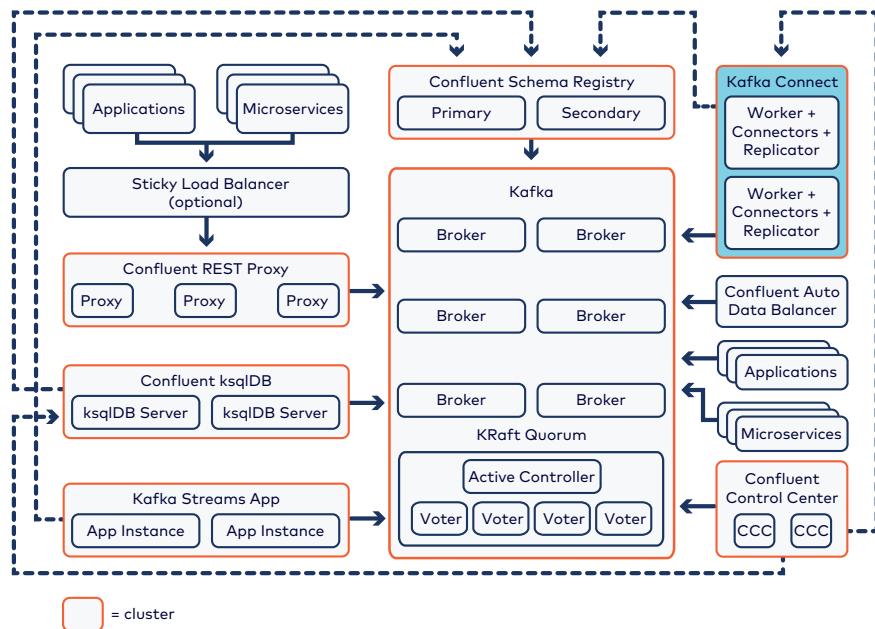
# 11b: What Does Confluent Advise for Deploying Kafka Connect in Production?

## Description

Best practices and capacity planning for Kafka Connect in production.

hitesh@datacouch.io

# Reference Architecture: Kafka Connect



# 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: [this reference](#).

## 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 multithreaded and so will benefit from multicore servers.

The more cores, the more threads can run simultaneously. You can configure the worker to take advantage of the multicore 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
```

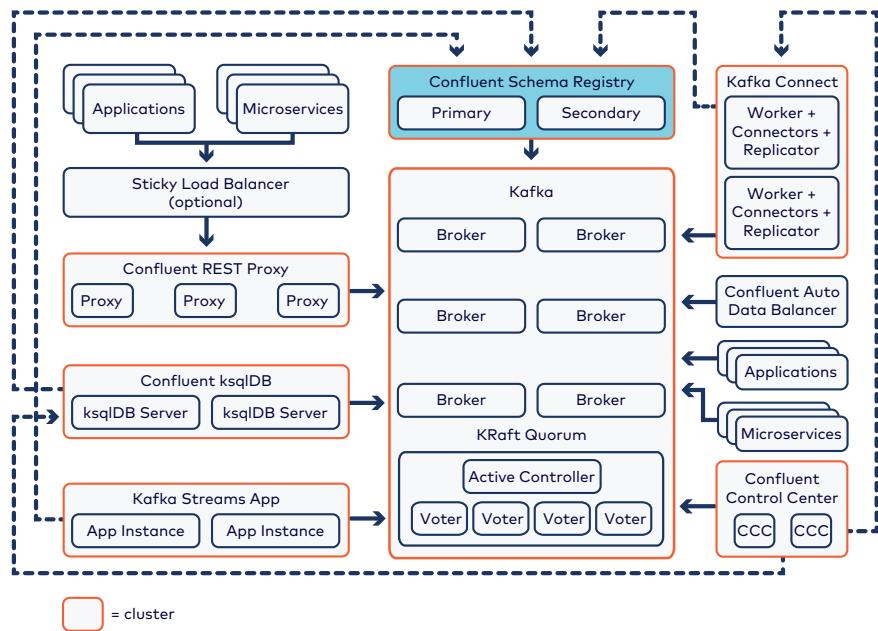
# 11c: What Does Confluent Advise for Deploying Schema Registry in Production?

## Description

Best practices and capacity planning for Schema Registry in production.

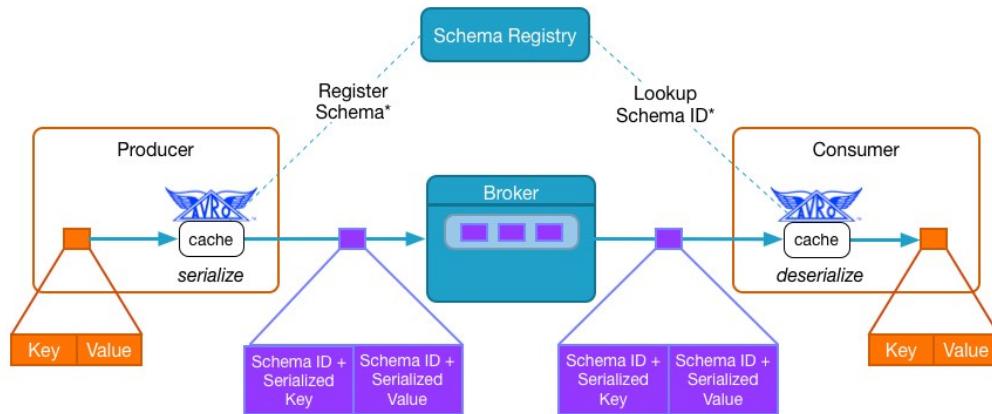
hitesh@datacouch.io

# 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 Developer 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 of the `--consumer_offsets` topic, where `offsets.topic.replication.factor` is enforced upon auto topic creation.

hitesh@datacouch.io

# 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 (basically virtualizing the `schema.registry.url` value across all clients).

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 by setting `kafkastore.bootstrap.servers` to a comma-separated string of Kafka endpoints.

	<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>
---	---

	<p>older versions relied on ZooKeeper if <code>kafkastore.connection.url</code> was set. If both this and <code>kafkastore.bootstrap.servers</code> 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: <a href="https://youtu.be/MmLezWRI3Ys?t=1776">https://youtu.be/MmLezWRI3Ys?t=1776</a></p>
---	--

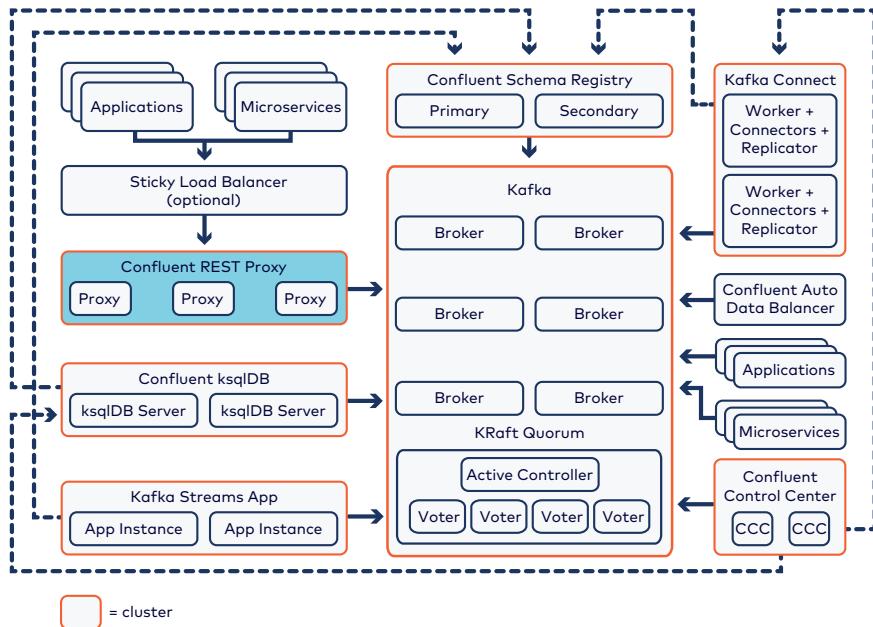
# 11d: What Does Confluent Advise for Deploying the REST Proxy in Production?

## Description

Best practices and capacity planning for the REST Proxy in production.

hitesh@datacouch.io

# Reference Architecture: Confluent REST Proxy



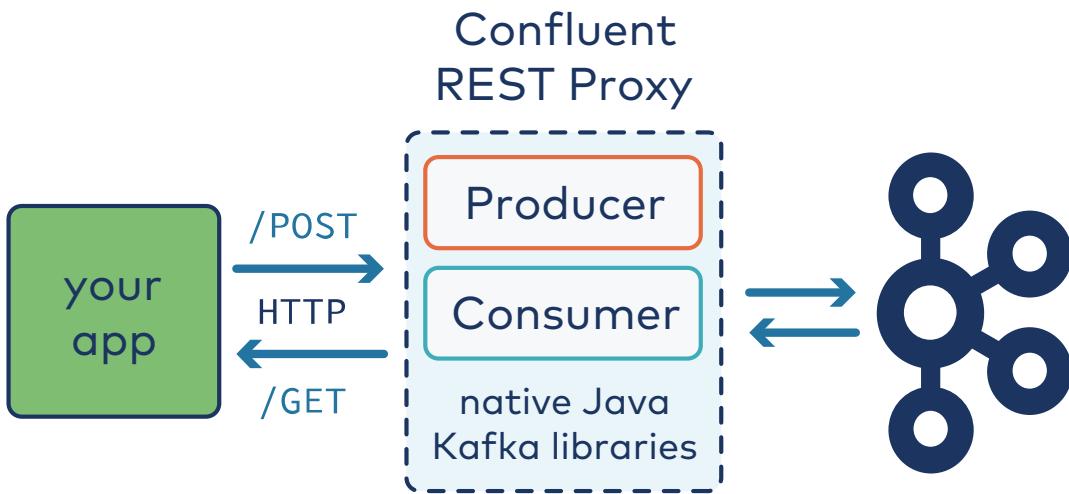
The Kafka REST Proxy is a part of Confluent Community. It provides a way to have producer and consumer style interaction with the brokers through HTTP calls. This is beneficial for:

- 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>.

# What is the Confluent REST Proxy?



Confluent Community includes a REST Proxy for Kafka. This allows **any language to access Kafka** via a REST interface over HTTP.

REST Proxy API reference: <https://docs.confluent.io/current/kafka-rest/api.html#crest-long-api-reference>

# 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.

	<p>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 <code>client.security.protocol</code> is configured to be either of <code>SSL</code>, <code>SASL_PLAIN</code>, or <code>SASL_SSL</code>. Confluent Enterprise Security Plugins are required to set fine-grained ACLs for individual clients. For more information, see <a href="#">this reference</a>.</p>
---	---

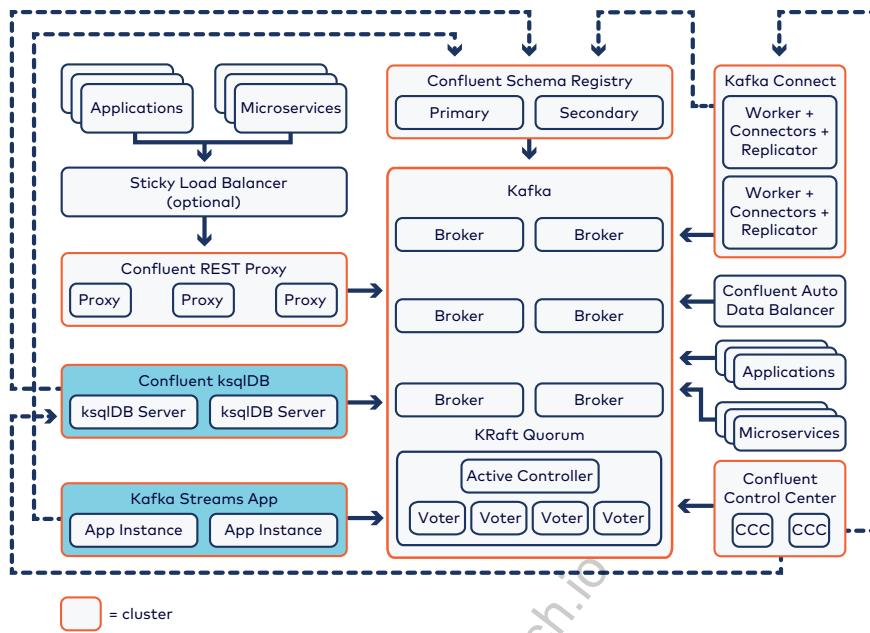
# 11e: What Does Confluent Advise for Deploying Kafka Streams and ksqlDB in Production?

## Description

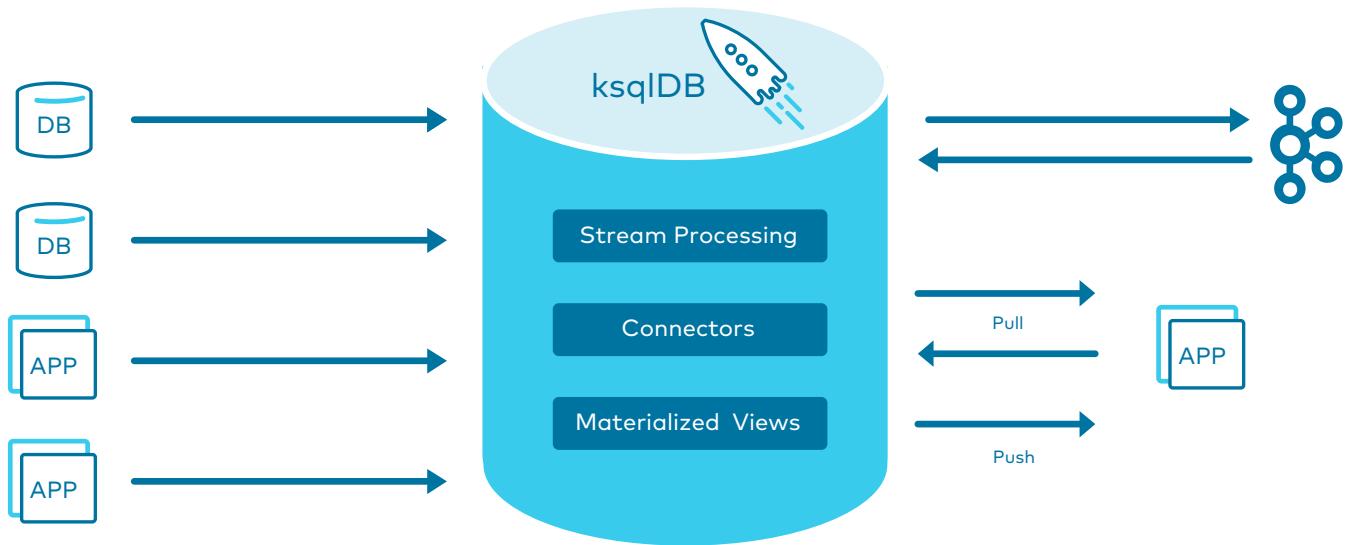
Best practices and capacity planning for Kafka Streams and ksqlDB in production.

hitesh@datacouch.io

# Kafka Reference Architecture: Kafka Streams and ksqlDB



# What is ksqlDB?



ksqlDB offers a single solution for collecting streams of data, enriching them, and serving queries on new derived streams and tables. It provides its own language, ksqlDB SQL, designed to be accessible to anyone who knows other flavors of SQL, and is meant to make the learning curve for stream processing less steep than otherwise.

ksqlDB is built on top of Kafka Streams and both are relevant here. With Kafka Streams...

- It is just a library that developers import into their code like any other library.
- Developers have to use Java or Scala.
- The syntax of the DSL (Domain Specific Language) is similar to the Streams API in Java or like Scala—the functional programming language.
- Instances of the application run with an `application.id`.
- Instances with the same `application.id` behave just like consumers in a consumer group.
- KStreams allows you to consume from a Kafka topic, process the data, and produce the results back to another Kafka topic.

# 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 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.

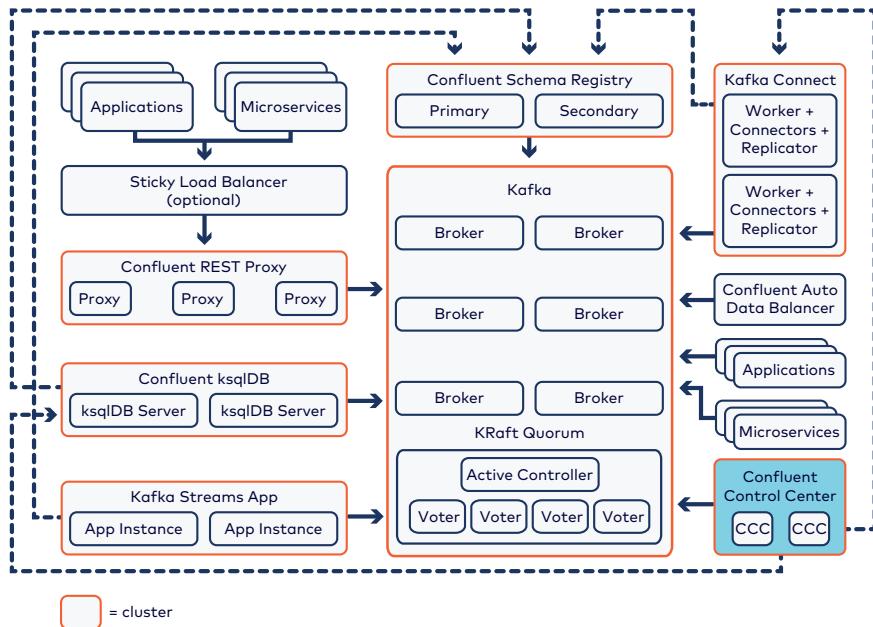
# 11f: What Does Confluent Advise for Deploying Control Center in Production?

## Description

Best practices and capacity planning for Confluent Control Center in production.

hitesh@datacouch.io

# Reference Architecture: Confluent Control Center

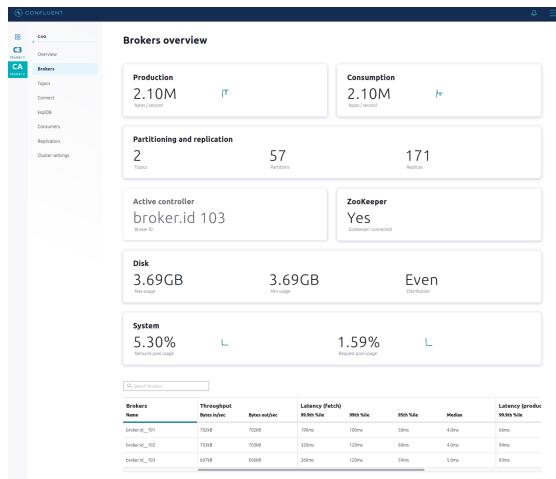


Each component in the architecture can be equipped with monitoring interceptors in its configuration to publish metrics to a Kafka cluster. Confluent Control Center connects to a primary Kafka cluster that holds all of these metrics.



Remember that CCC requires an enterprise license.

# Highly Available Confluent Control Center



- 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`).

# Conclusion



**CONFLUENT  
Global Education**

hitesh@datacouch.io

# Course Contents



Now that you have completed this course, you should have the skills to:

- Describe how Kafka brokers, producers, and consumers work
- Describe how replication works within the cluster
- Understand hardware and runtime configuration options
- Monitor and administer your Kafka cluster
- Integrate Kafka with external systems using Kafka Connect
- Design a Kafka cluster for high availability & fault tolerance

hitesh@datacouch.io

# 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®
- Managing Data in Motion with Confluent Cloud



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

- 
- **Confluent Developer Skills for Building Apache Kafka®** has these objectives:
    - Write Producers and Consumers to send data to and read data from Apache Kafka
    - Create schemas, describe schema evolution, and integrate with Confluent Schema Registry
    - Integrate Kafka with external systems using Kafka Connect
    - Write streaming applications with Kafka Streams & ksqlDB
    - Describe common issues faced by Kafka developers and some ways to troubleshoot them
    - Make design decisions about acks, keys, partitions, batching, replication, and retention policies
  - **Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB** has these objectives:
    - 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

- **Confluent Advanced Skills for Optimizing Apache Kafka** has these objectives:

- 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

hitesh@datacouch.io

# Confluent Certified Developer for Apache Kafka

**Duration:** 90 minutes

**Qualifications:** Solid understanding of Apache Kafka and Confluent products, and 6-to-9 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
  - Multiple choice questions
  - 90 minutes
  - Designed to validate professionals with a minimum of 6-to-9 months hands-on experience
  - Remotely proctored on your computer
  - Available globally in English

# 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

# We Appreciate Your Feedback!



Please complete the course survey now.

---

Your instructor will give you details on how to access the survey

hitesh@datacouch.io

**Thank You!**

*hitesh@datacouch.io*

# **Appendix: Confluent Technical Fundamentals of Apache Kafka® Content**



**CONFLUENT  
Global Education**

hitesh@dataeducation

# Module Overview



This section contains 5 lessons - the content lessons from the Fundamentals prerequisite:

Lessons of Presentation:

1. Getting Started
2. How are Messages Organized?
3. How Do I Scale and Do More Things With My Data?
4. What's Going On Inside Kafka?
5. Recapping and Going Further

hitesh@datacouch.io

# 1: Getting Started



**CONFLUENT  
Global Education**

hitesh@datacouch.io

# Why Kafka?

In a nutshell...

Kafka is good for	Kafka is not meant for
<ul style="list-style-type: none"><li>• data in motion</li><li>• real-time processing</li></ul>	<ul style="list-style-type: none"><li>• batch processing</li><li>• archiving data</li></ul>

hitesh@datacouch.io

# One Example Use Case: Ordering Food

Suppose we are building a system for a restaurant chain:

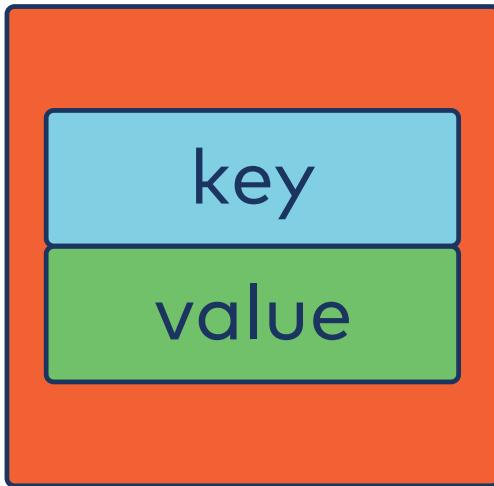
- customers order food via an app - mobile or kiosk
- staff receive orders to fulfill in real-time
- management tracks inventory based on orders

We will build up some of the fundamental details of Kafka and use this example.

hitesh@datacouch.io

# Messages

The atomic unit of Kafka is a **message** or **record** or **event**



---

This treatment is deliberately simple: just key and value. You can specify other components in specifying a message; there's more in the Developer and Administrator training.

It is possible to have messages without keys; this course will assume all messages have keys specified.

# Topics

Messages are organized in logical groups called **topics**.

Example topics:

- **orders**
  - menu items
  - customers
  - restaurants
- 

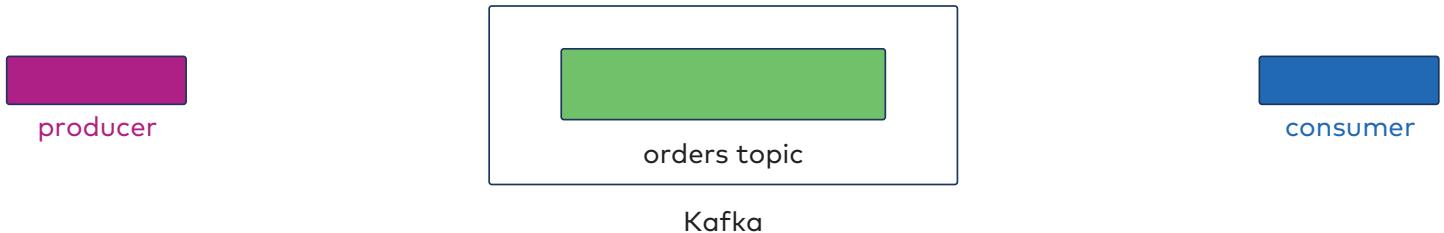
The topics listed go along with the example use case from a few slides back.

The topic **orders** is bold because we'll focus on that one specifically.

hitesh@datacouch.io

# Three Basic Components

Let's start simple - with an **orders** topic in place in Kafka, a producer, and a consumer:



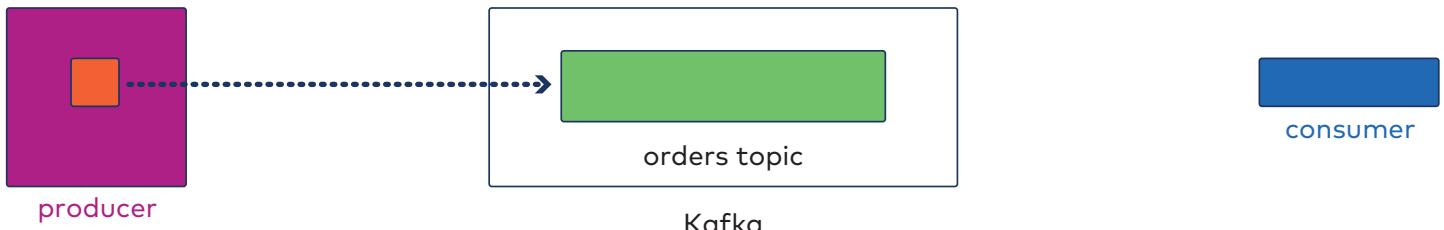
---

Here, we just show three components at a high level. The next three slides go into each in more detail.

We'll add even more detail in the lessons to come and wrap up with more detailed views of these illustrations.

# Life Cycle of a Message: Producing

A **producer** prepares messages and publishes them to Kafka.



hitesh@datacouch.io

# Life Cycle of a Message: Kafka

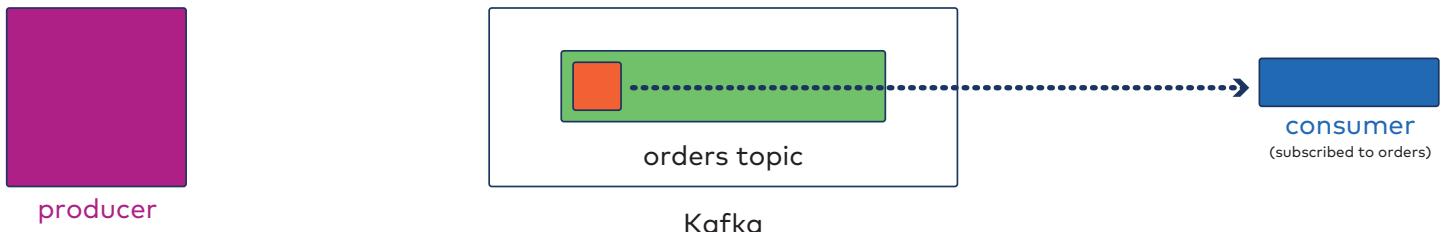
Produced messages live in Kafka, organized by topic.



hitesh@datacouch.io

# Life Cycle of a Message: Consumption

Consumers subscribe to topics in Kafka and poll for new messages.



---

When we set up consumers, we subscribe to topics to read from.

Note that the message is still shown in the topic in Kafka even though it is shown in the consumer. This is correct. Consuming a message does not remove it from Kafka.

## 2: How are Messages Organized?



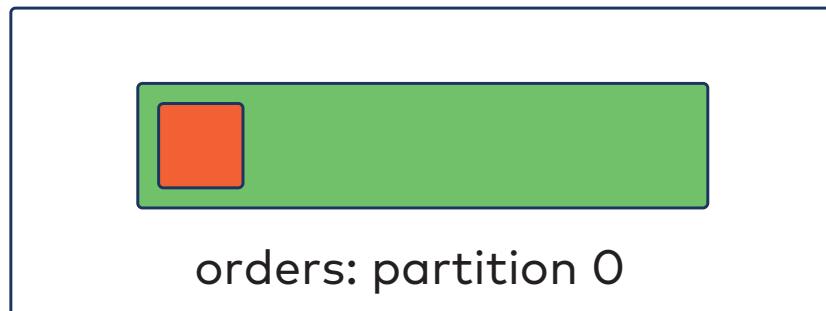
**CONFLUENT  
Global Education**

hitesh@datacouch.io

# Topics and Partitions

Topics are broken down into **partitions**

Simplest case: Topic with one partition



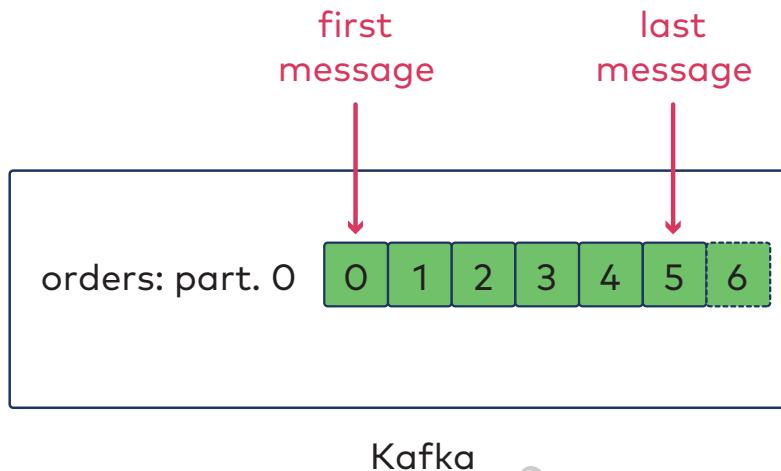
Kafka

---

The presentation is meant to be only about logical organization at this point, and the messages in a partition are a (logical) subset of the messages in a topic. A partition is also a *physical* grouping of messages; more on the physical later.

# Offsets—in Kafka

- Each message in a partition has an **offset**
- Starting from 0

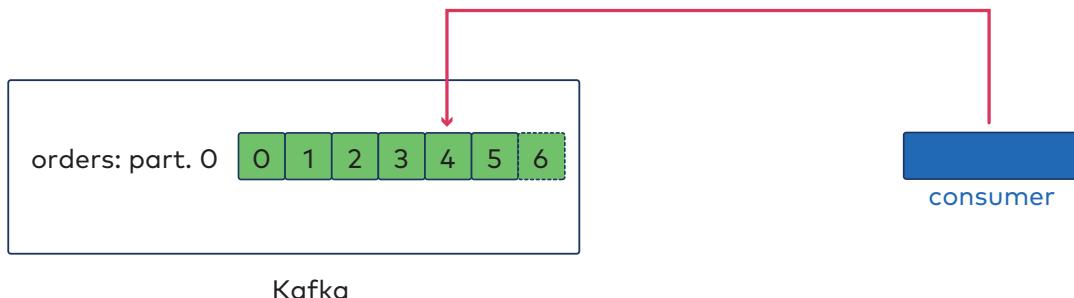


A different way of thinking of a message's offset is "how many messages were written to this partition before this message?"

Here we point out the offset of the last message. The offset where the next message will be written is illustrated too; the hands-on activity will show this offset being reported a Kafka command line tool.

# Offsets—Consumer Offsets

- Consumers track where they will read next via a **consumer offset**



In this picture, the consumer has last read the message at offset 3.

For now, let's say our consumer is assigned to the **orders** topic and it has one partition, and in turn our consumer must be assigned to that one partition. A consumer offset is:

- per consumer
- per partition

But there is only one of each so far, so there is one offset. Multiple consumers and multiple consumers will come in the next lesson.

## Check Your Knowledge!

Try a [quick quiz on Lessons 1 and 2.](#)



hitesh@datacouch.io

### 3: How Do I Scale and Do More Things With My Data?



**CONFLUENT**  
**Global Education**

hitesh@datacouch.io

# Scaling Up...

So far, we have seen...

- one partition
- one consumer

In practice...

- multiple consumers in a consumer group
- multiple consumer groups
- multiple partitions in a topic

---

In this lesson, we'll expand on the last and look into having multiple consumers or partitions.

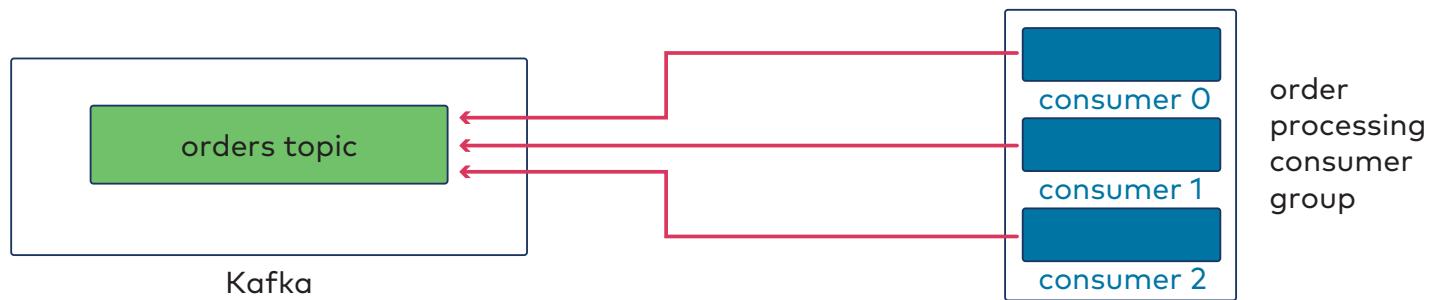
hitesh@datacouch.io

# Consumer Groups

Consumers exist in **consumer groups**

Consumers in a group:

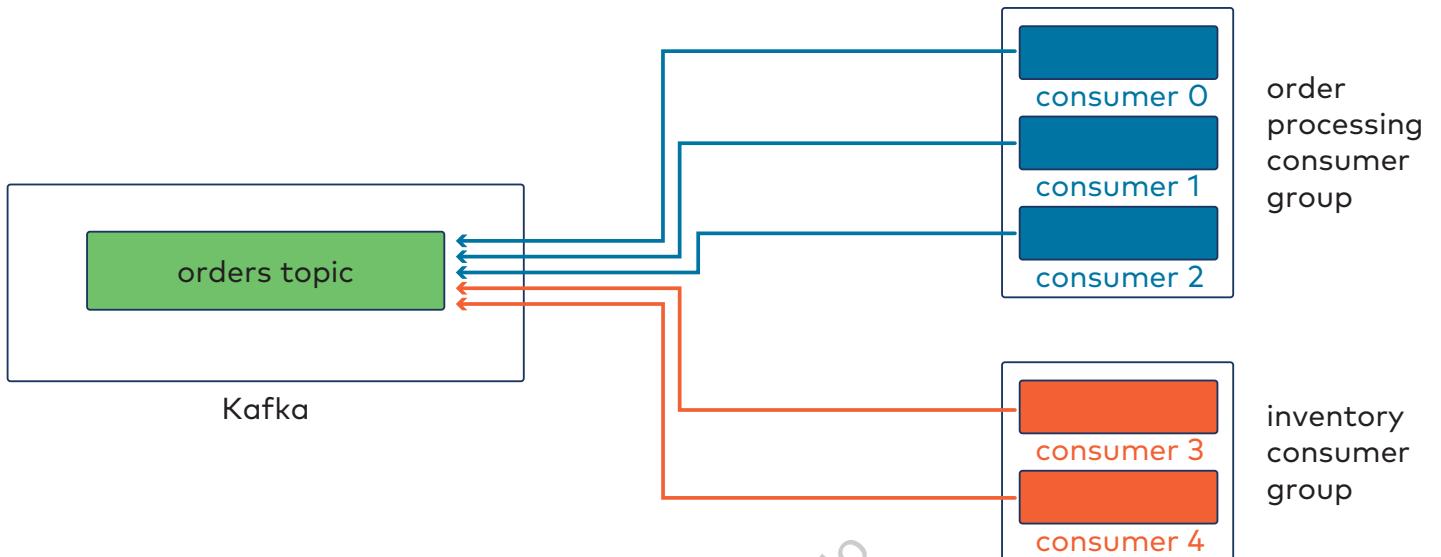
- **same** application
- **different** data



Technically, it is recommended but not required consumers be in groups. We'll always model consumers in groups.

# Multiple Consumption

Could have multiple groups using the same data...

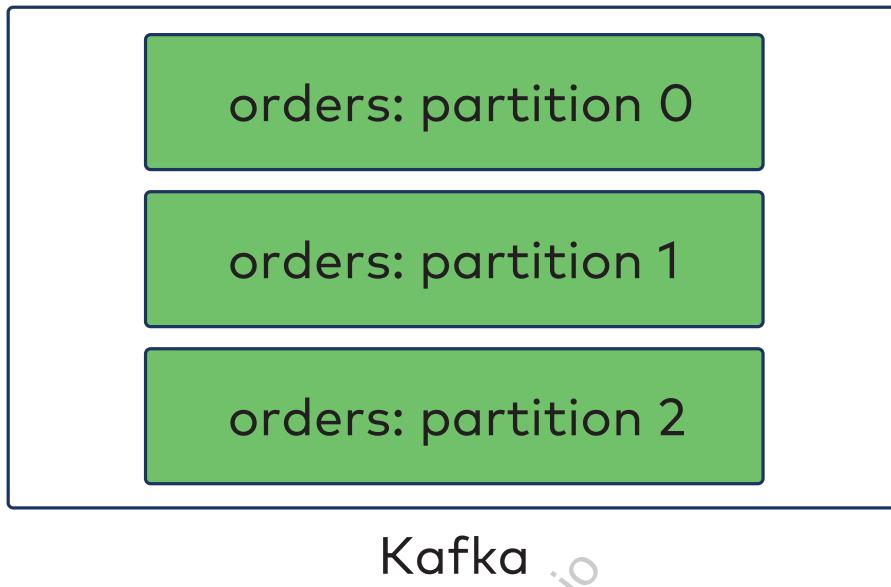


When a consumer reads a message, that does not remove the message. Different consumers could read the same message.

All of the consumers in a group are doing the same task. We see here our order processing group as before, but we add a second group of consumers: they go through orders (with less urgency) to tally what has been ordered and help alert a restaurant which ingredients or supplies should be restocked.

# Multiple Partitions

In practice, we want topics to have multiple partitions.

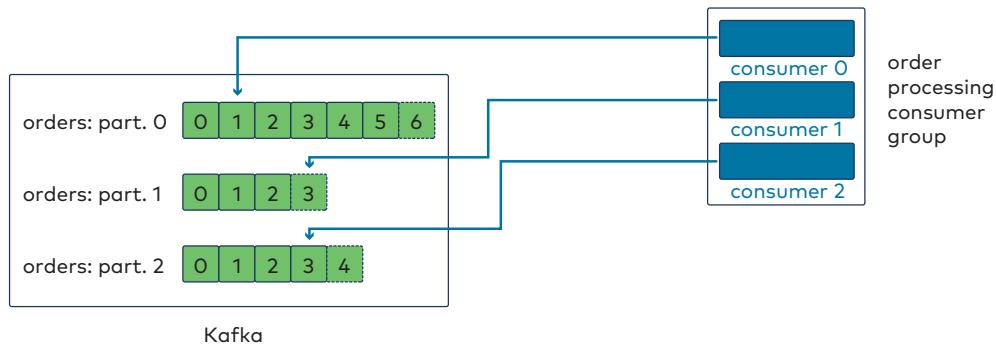


Before, we said a partition was a subset of the messages in a topic, but we only saw one partition. Here is where we first see a topic broken up into multiple partitions.

# Consuming from Multiple Partitions

Now our consumers can consume in parallel:

- Consumers subscribed to a topic are assigned partitions
- Group covers all partitions
- Each consumer has an offset for each partition

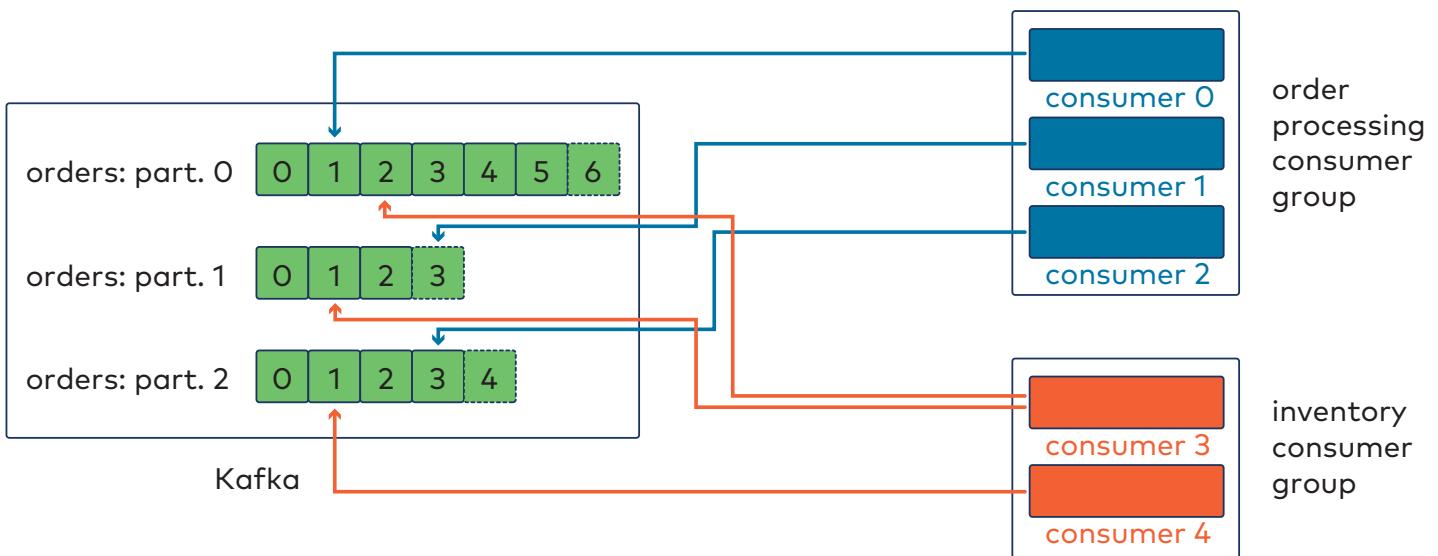


Now that we have multiple partitions, different consumers could read from different partitions all at the same time.

Kafka handles the assignment of consumers to partitions; all users need to do is subscribe to topics. There are configuration settings and details; more on this in the Administrator and Developer classes.

Each consumer could be reading from each partition at a different location, so consumer offsets are per consumer and per partition. The picture illustrates three consumers all at different locations in their assigned partitions.

# Expanding the Last Picture



Here we bring back the second consumer group from a few slides ago. We can see these consumers assigned to partitions. The group as a whole needs to read all messages in all partitions and we have two consumers but three partitions, so one consumer has to handle two partitions this time.

Note, also, that different consumers reading from the same partition don't have to be at the same place, as shown in this picture.

# How Do Messages Get Partitioned?

- **Producers** decide which messages go to which partition
  - Partitions are indexed from 0 to `numberOfPartitions - 1`
  - Default partitioner: `partitionIndex = hash(key) % numberOfPartitions`
- 

It is the producer that decided which messages go to which partition of a topic. The slide gives the formula used by default - and which is used by the console producer you'll use in the hands-on exercise.

Developers may specify a partitioner in producer code.

hitesh@datacouch.io

# Scaling is Easy!

Say you want to

- Increase the number of consumers in a group during a busy season
- Decrease the number of consumers in a group when things are slow
- Increase the number of partitions for a topic

When you do, Kafka *automatically* redistributes the assignments of consumers to partitions!



More on how all of this works in both our Developer and Administrator training!

---

You can change some factors about your Kafka deployment after you've started using it. When you change what's on the slide, it will cause Kafka to change which consumers work with which partitions in what is called a *rebalance*. The details of how this work are beyond the scope of this course but get significant attention in the Developer and Administrator classes.

## 4: What's Going On Inside Kafka?



**CONFLUENT  
Global Education**

hitesh@datacouch.io

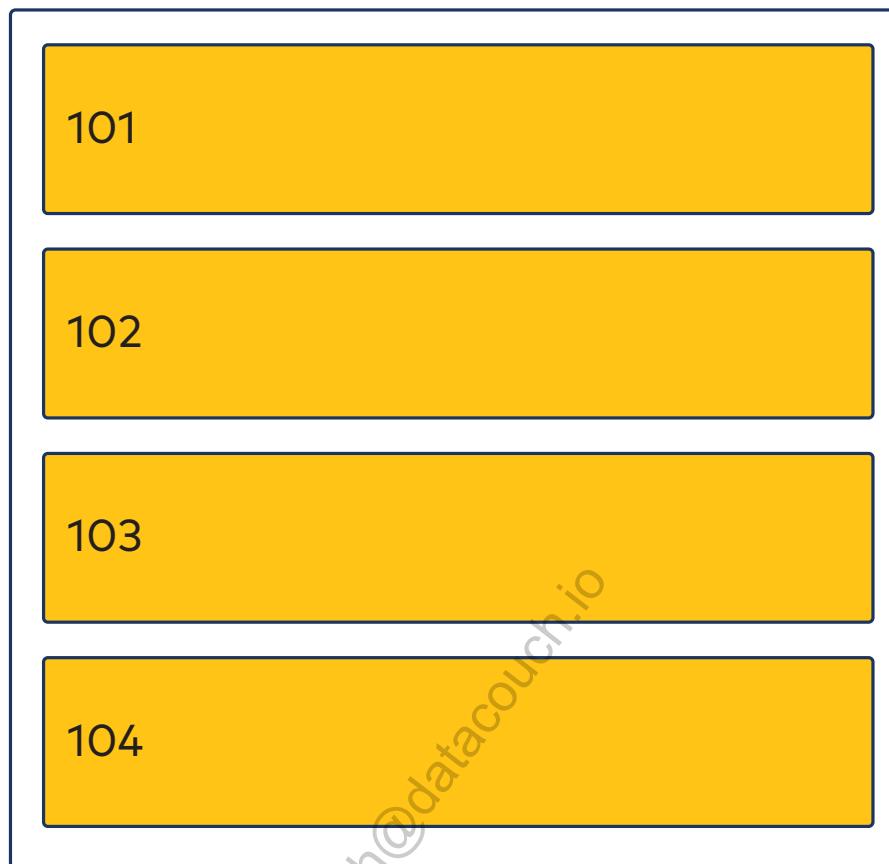
## Going Deeper...

Now let's learn about some more details about a Kafka cluster, especially *physical* things...

hitesh@datacouch.io

# Brokers

A Kafka cluster consists of multiple **brokers**



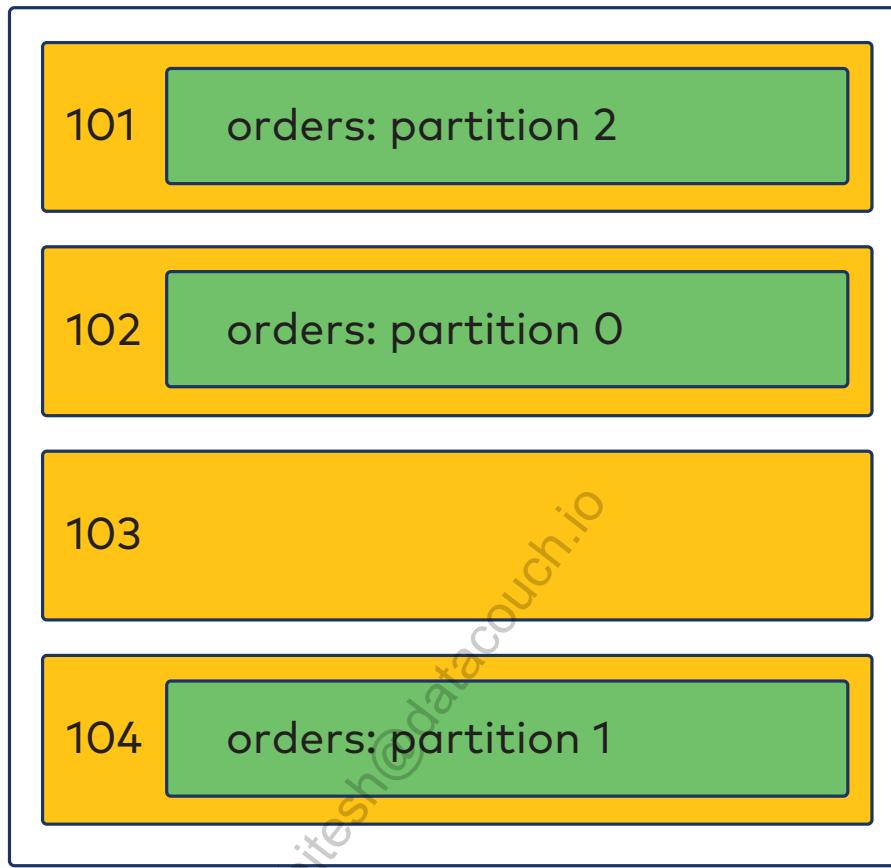
Brokers could be physical servers, but could also be VMs, Docker containers, etc.

The orange boxes represent brokers and the numbers are broker IDs.

# Partitions & Brokers

Partitions are really **physical** groupings of the messages in topics.

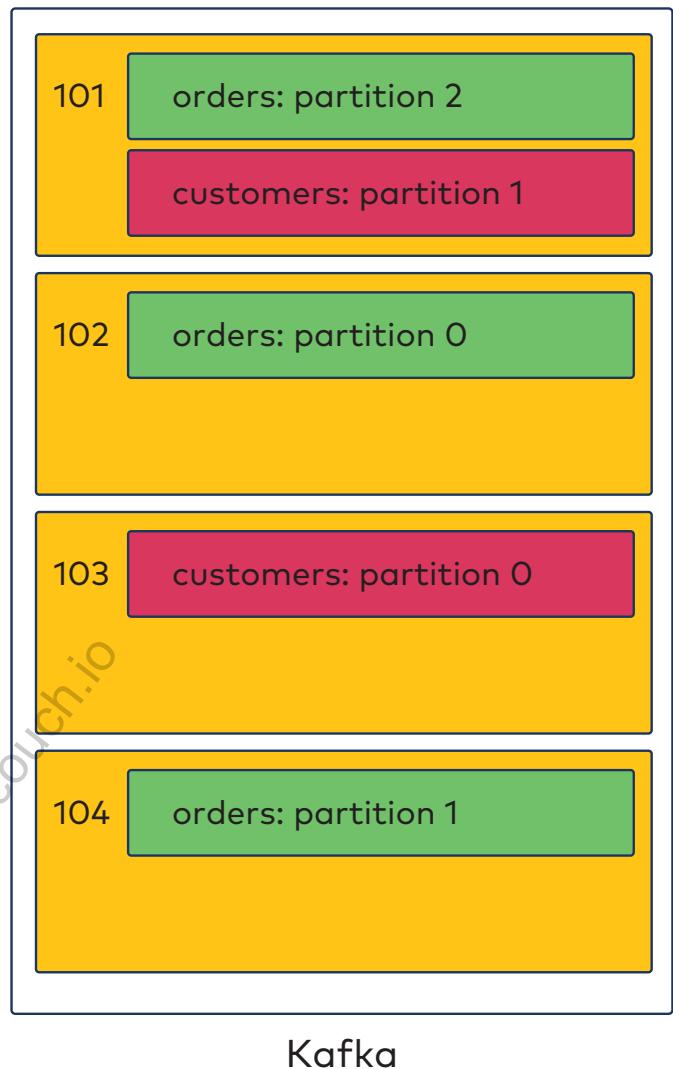
Partitions are stored on brokers.



## Partitions & Brokers (2)

The number of partitions is a topic setting.

Kafka decides how partitions get distributed across brokers.

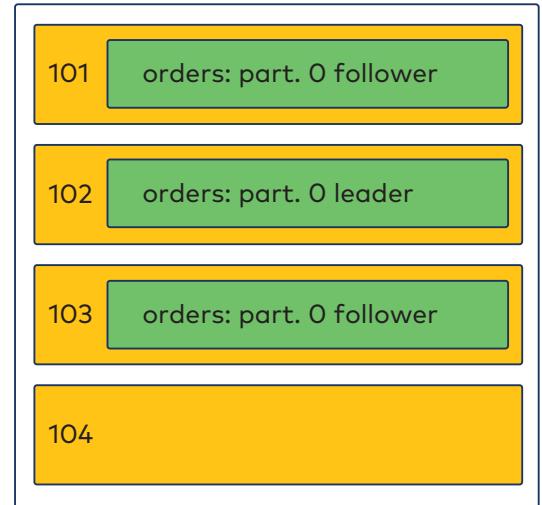


---

Here we see a second topic with a different number of partitions added to the illustration.

# What if a Broker Goes Down?

- Want *high availability* of data in partitions
- Achieved via **replication**
- Writes are reads go to **leader** replica
- **Follower** replicas keep backup copies of the leader
- If leader dies, a follower becomes the leader



Kafka

Note that this is a very simple treatment and replication is covered in much more detail in the Developer and Administrator courses.

# Serialization and Deserialization

- Kafka stores messages as byte arrays
  - Producers must **serialize** messages
  - Consumers must **deserialize** messages
- 

In developing custom producers and consumers, developers must specify the serializers and deserializers. There are also tools like Avro and Protobuf that can be used for complex data types.

hitesh@datacouch.io

# Immutable Messages

- Messages are **immutable**
- Once written, we cannot change anything about them

hitesh@datacouch.io

# ...But We Don't Keep Messages Forever...

Control which messages stay in Kafka via a **retention policy**:

Policy	Deletion	Compaction																																																																						
Idea	Remove messages older than a certain age (default 7 days)	Keep only the latest value for each key																																																																						
Before	offset <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>a</td><td>b</td><td>b</td><td>a</td><td>a</td></tr><tr><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr></table> key <table border="1"><tr><td>a</td><td>b</td><td>b</td><td>a</td><td>a</td></tr><tr><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr></table> age in days <table border="1"><tr><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr><tr><td>6</td><td>5</td><td>2</td><td></td><td></td></tr></table>	0	1	2	3	4	a	b	b	a	a	12	10	6	5	2	a	b	b	a	a	12	10	6	5	2	12	10	6	5	2	6	5	2			offset <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>a</td><td>b</td><td>b</td><td>a</td><td>a</td></tr><tr><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr></table> key <table border="1"><tr><td>a</td><td>b</td><td>b</td><td>a</td><td>a</td></tr><tr><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr></table> age in days <table border="1"><tr><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr><tr><td>6</td><td>5</td><td>2</td><td></td><td></td></tr></table>	0	1	2	3	4	a	b	b	a	a	12	10	6	5	2	a	b	b	a	a	12	10	6	5	2	12	10	6	5	2	6	5	2		
0	1	2	3	4																																																																				
a	b	b	a	a																																																																				
12	10	6	5	2																																																																				
a	b	b	a	a																																																																				
12	10	6	5	2																																																																				
12	10	6	5	2																																																																				
6	5	2																																																																						
0	1	2	3	4																																																																				
a	b	b	a	a																																																																				
12	10	6	5	2																																																																				
a	b	b	a	a																																																																				
12	10	6	5	2																																																																				
12	10	6	5	2																																																																				
6	5	2																																																																						
After	offset <table border="1"><tr><td>2</td><td>3</td><td>4</td></tr><tr><td>6</td><td>5</td><td>2</td></tr></table> age in days <table border="1"><tr><td>6</td><td>5</td><td>2</td></tr><tr><td></td><td></td><td></td></tr></table>	2	3	4	6	5	2	6	5	2				offset <table border="1"><tr><td>2</td><td>4</td></tr><tr><td>b</td><td>a</td></tr></table> key <table border="1"><tr><td>b</td><td>a</td></tr><tr><td></td><td></td></tr></table>	2	4	b	a	b	a																																																				
2	3	4																																																																						
6	5	2																																																																						
6	5	2																																																																						
2	4																																																																							
b	a																																																																							
b	a																																																																							



Partitions are divided into segments, which affect both retention policies.

Note that this is a very simple treatment and these policies are covered in much more detail in the Developer and Administrator courses. In particular, the impact of segments matters. Deletion is per segment - but the details of segments comes in the other courses.

Maybe, for a deletion use case, we might keep orders around for up to a week to track trends on what people ordered in the last week.

Maybe, for a compaction use case, we might have some sort of greeting for a returning customer like "last time, you ordered... would you like to order this again?"

But, above all, these policies are about smart use of storage.

## Check Your Knowledge!

Try a [quick quiz on Lessons 3 and 4.](#)



hitesh@datacouch.io

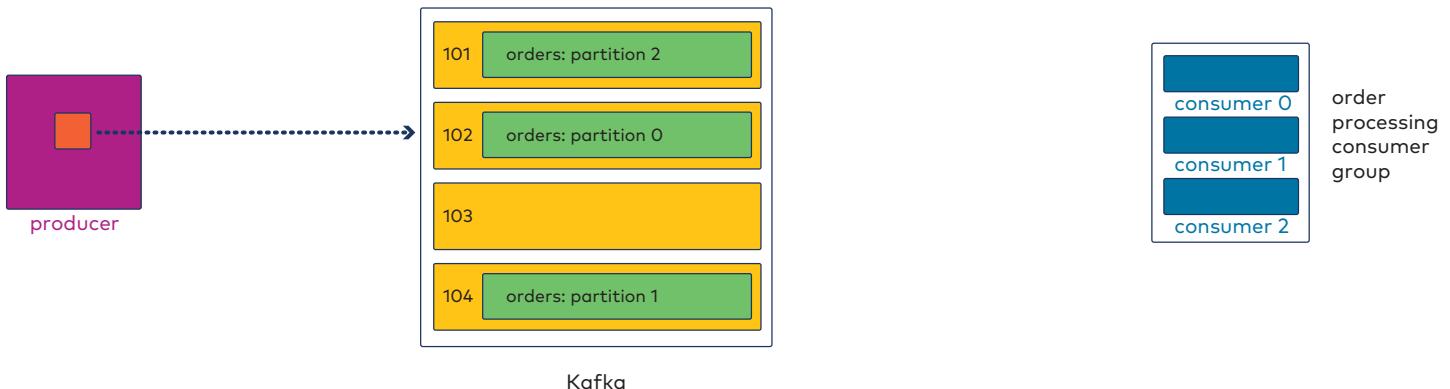
## 5: Recapping and Going Further



**CONFLUENT  
Global Education**

hitesh@datacouch.io

# Life Cycle of a Message: Producing



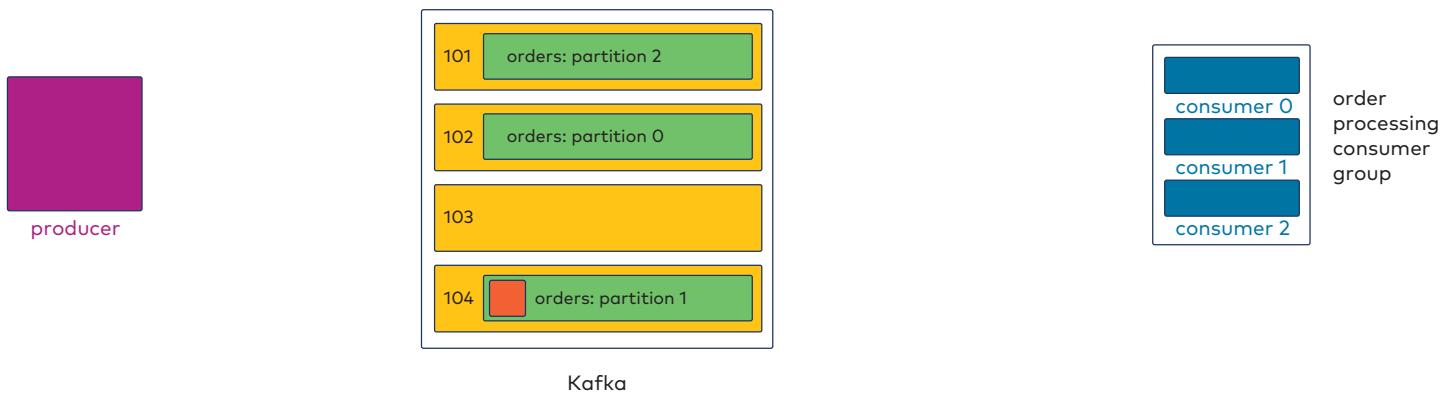
- Producers serialize and partition messages
- Producers send messages
  - ...in batches - can be configured for throughput and latency desires

Let's summarize what we've learned about the life cycle of a message... in the context of our running example. We work with a setup similar to what we had in the first lesson, with some of the details from later lessons added.

We start here with a message on a producer being sent...

# Life Cycle of a Message: Kafka

Produced messages live in Kafka, organized by topic.

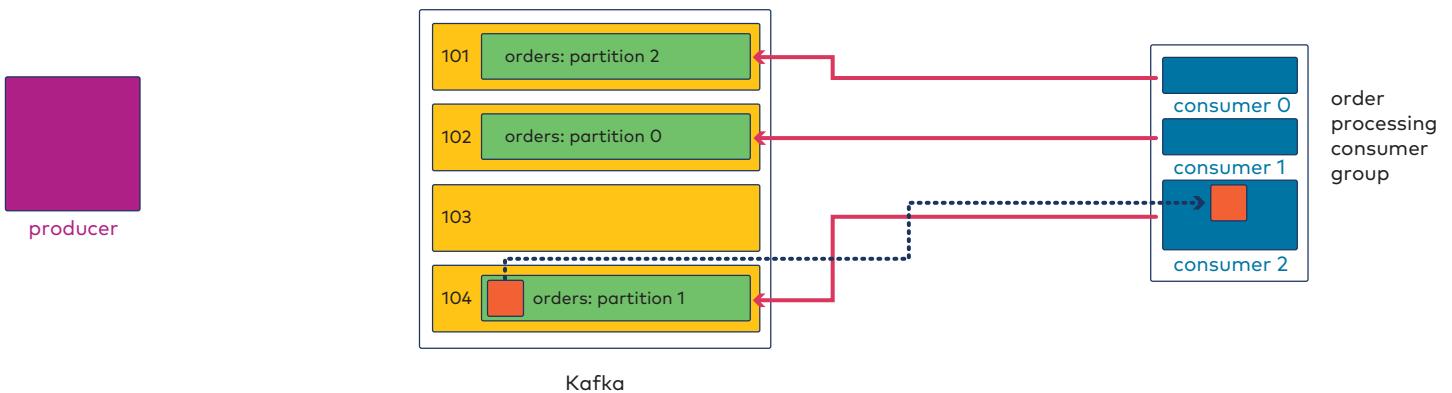


- Kafka consists of brokers
- Brokers contain partitions, which contain messages
- Brokers handle retention and replication

Now our message makes it to Kafka, specifically to a partition of a topic, and that partition lives on a particular broker.

# Life Cycle of a Message: Consumption

Consumers subscribe to topics in Kafka and poll for new messages.



- Consumers operate in groups
- Consumers subscribe to topics, are assigned partitions of those topics
- Consumers poll for messages in partitions at consumer offsets
  - ...and fetch in batches - can be configured for throughput and latency desires

---

Now we see a group of consumers subscribed to the orders topic and see a message being read by one of those consumers.

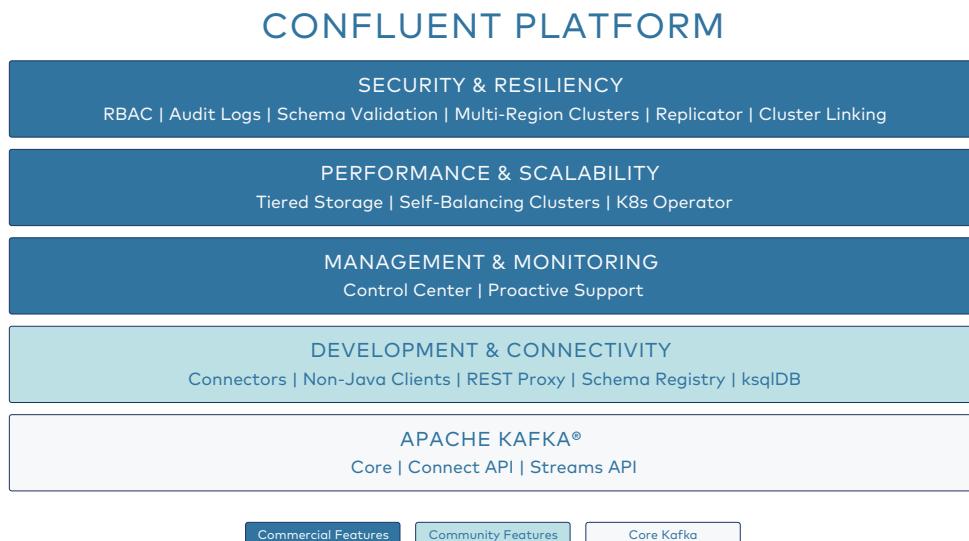
# A Step Beyond Fundamentals: Other Components

We've addressed some aspects of Core Kafka in this course. Some other topics you may want to learn about include:

- **Kafka Connect** - a tool that helps you copy data to Kafka from other systems and vice-versa
- **Kafka Streams** - a layer on top of the Producer and Consumer APIs that allows for stream processing
- **Confluent ksqlDB** - a tool for stream processing using a more-accessible SQL-like syntax, among other things
- **Confluent Schema Registry** - a tool for managing schemas, guiding schema evolution, and enforcing data integrity

You can learn more about these topics in our Confluent Developer Skills for Building Apache Kafka® and Apache Kafka® Administration by Confluent courses.

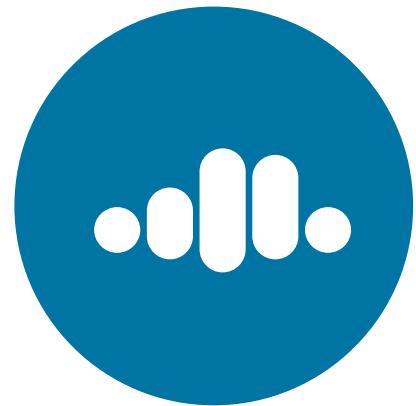
# What Does Confluent Platform Add to Kafka?



Everything we've discussed in this course is part of core Apache Kafka. Confluent Platform adds additional features beyond the core. The top two boxes in the medium shade of blue are paid features; the next two in the teal shade of blue are free features.

# Confluent Cloud

- Can deploy CP as self-managed software but...
- Confluent Cloud = **fully-managed** deployment of CP
  - Many administrative tasks done for you
- Confluent Cloud available on
  - AWS
  - Google Cloud Platform
  - Microsoft Azure



---

Everything we've learned about in this course is independent of platform, but CP may be deployed in a self-managed way or via Confluent Cloud, where our team handles many management tasks for you.

hitesh@datacoach.io

# Your Next Steps

Labs:

1. Complete interactive lab on seeing console producers and consumers in action.

→ [Short Confluent Cloud version](#)

→ Gitpod version: [More involved version using Gitpod](#)



2. Work though other Critical Thinking Challenge Exercises.

→ [On the web](#)

→ Solutions on the web too!

Critical  
Thinking:



3. Enroll in and complete one of these courses, as suits your role:

◦ Apache Kafka® Administration by Confluent

◦ Confluent Developer Skills for Building Apache Kafka®

hitesh@datacouch.io

# Thank You

Thank you for attending the course!

hitesh@datacouch.io

# Appendix: Additional Content



CONFLUENT  
**Global Education**

hitesh@datacouch.io

# Overview

This appendix contains a few additional lessons. These lessons are for additional information for you, but are not designed the same as the rest; namely, they do not have activities or labs to reinforce the content like the rest.

Some lessons that were part of modules of a previous version of this course. Some are lessons taken from another course.

hitesh@datacouch.io

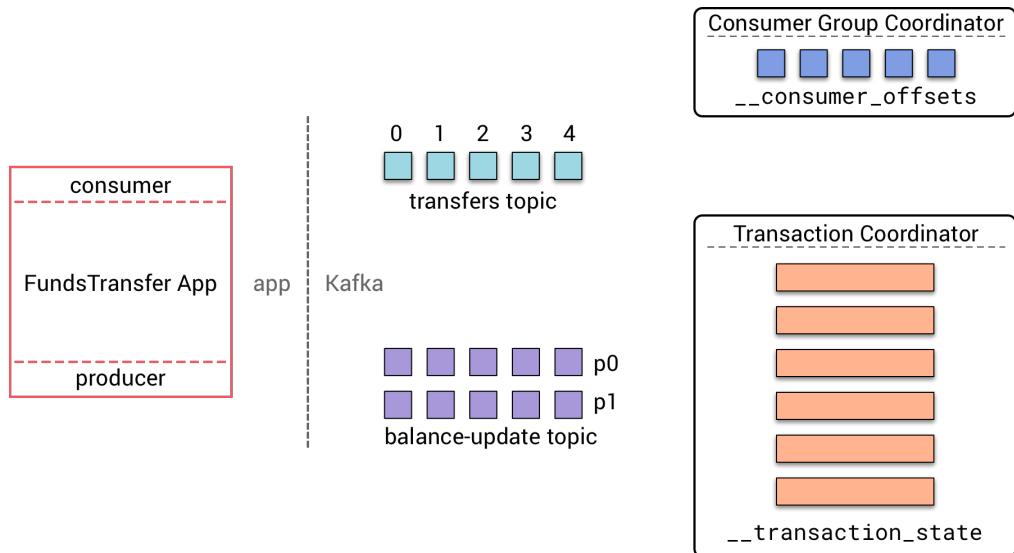
# Appendix A: Detailed Transactions Demo

## Description

This section presents a more detailed demo of a consume-process-produce application that uses transactions.

hitesh@datacouch.io

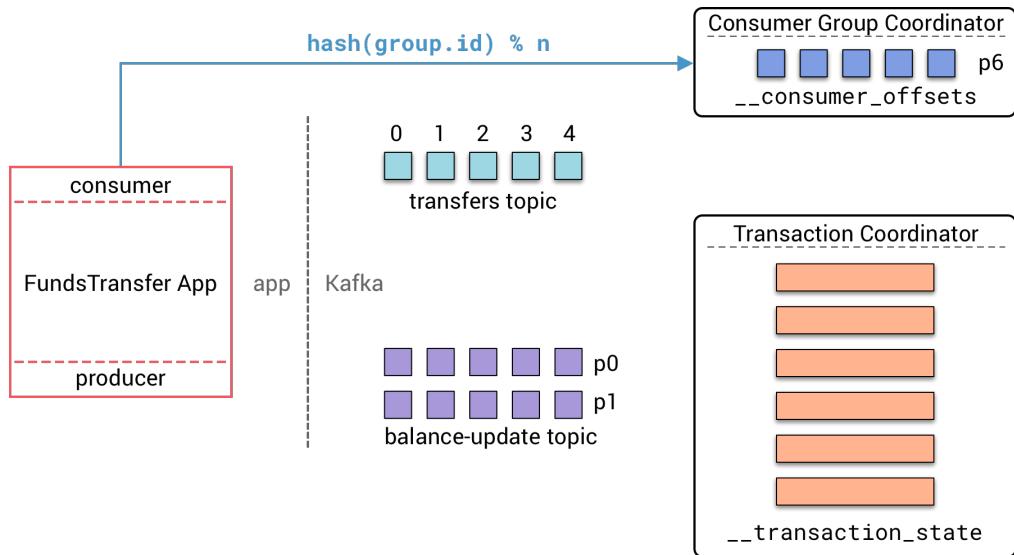
## Transactions (1/14)



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.

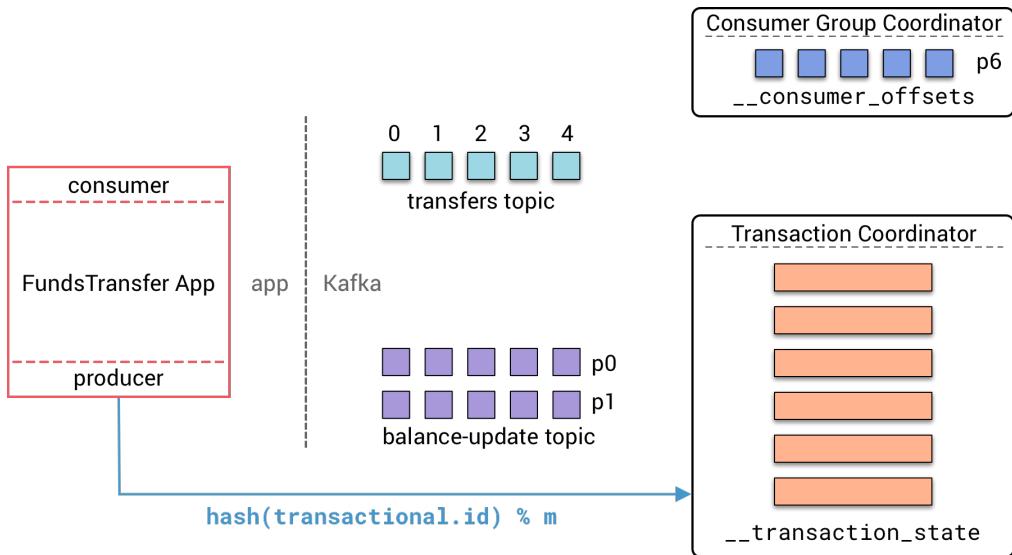
A Transaction Coordinator is a module that is available on any Broker. The Transaction Coordinator is responsible for managing the lifecycle of a transaction in the "Transaction Log"—the internal Kafka Topic **\_\_transaction\_state** partitioned by **transactional.id**. The Broker that acts as the Transaction 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)



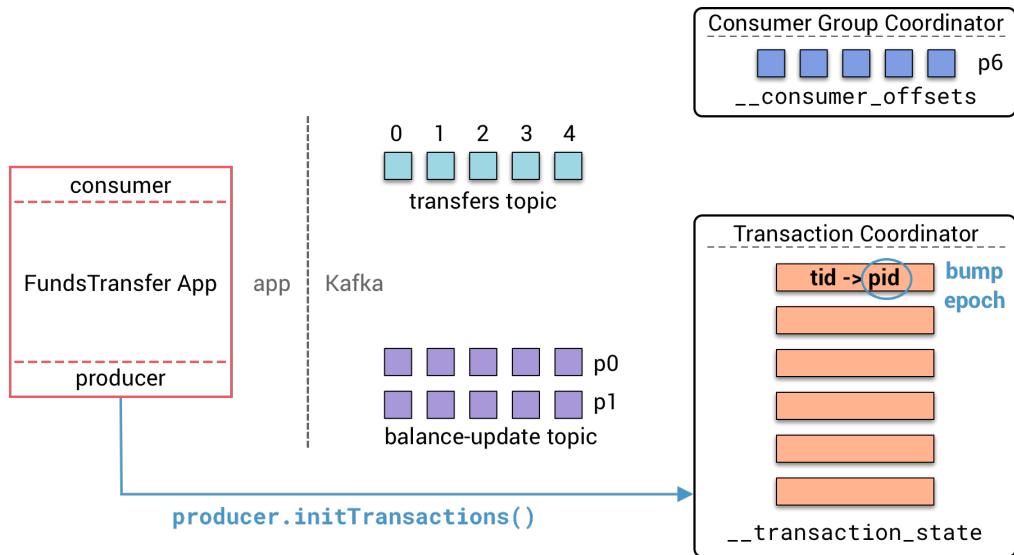
The consumer and producer are initialized before stream processing is started. Here we see the consumer subscribe to the "transfers" topic and identify its Consumer Group Coordinator using `hash(group.id) % n`, where `n` is the number of partitions of the consumer offsets topic (Default: 50). Here, the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

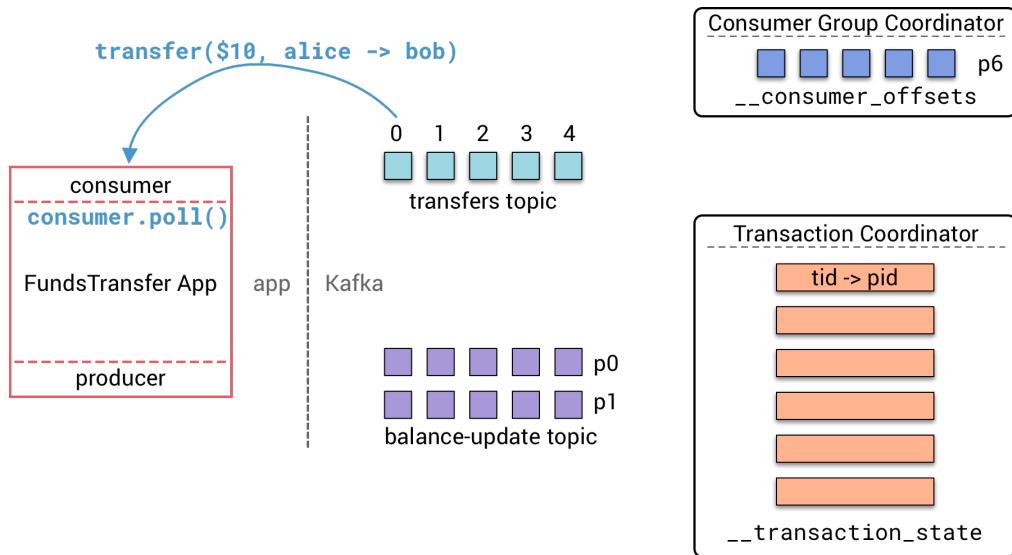
## Transactions - Initialize (4/14)



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**.

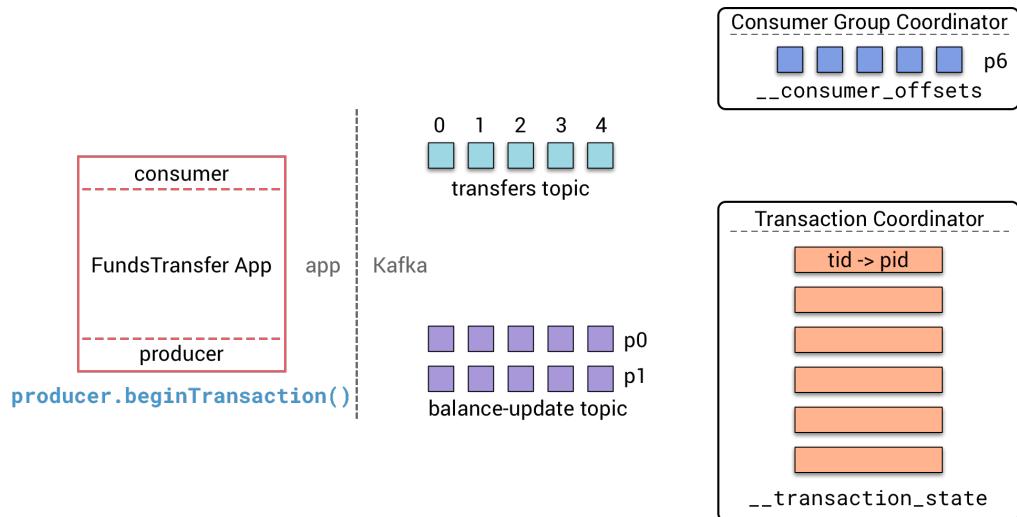
The epoch is an internal piece of metadata stored for every **transactional.id**. Once the epoch is bumped, any producers with same **transactional.id** and an older epoch are considered zombies and are fenced off and future transactional writes from those producers are rejected. This enables reliability 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)



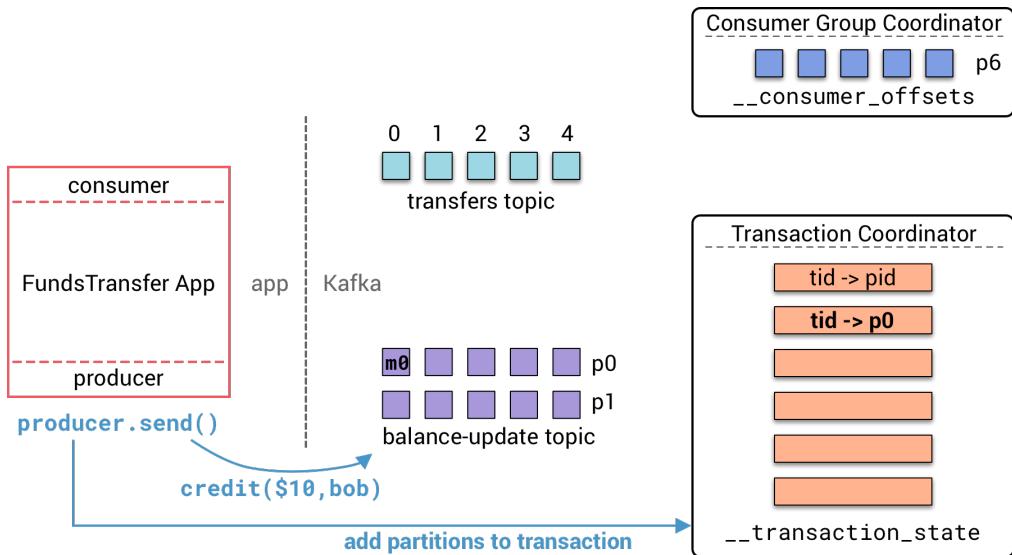
The Consumer polls for messages from the input Topic. Here, the consumer reads an event that transfers \$10 from Alice to Bob. The goal of the FundsTransfer app is to transactionally write events to the "balance-update" topic that credits Bob with \$10 and debits \$10 from Alice.

## Transactions - Begin Transaction (6/14)



The Producer begins the transaction.

## Transactions - Send (7/14)

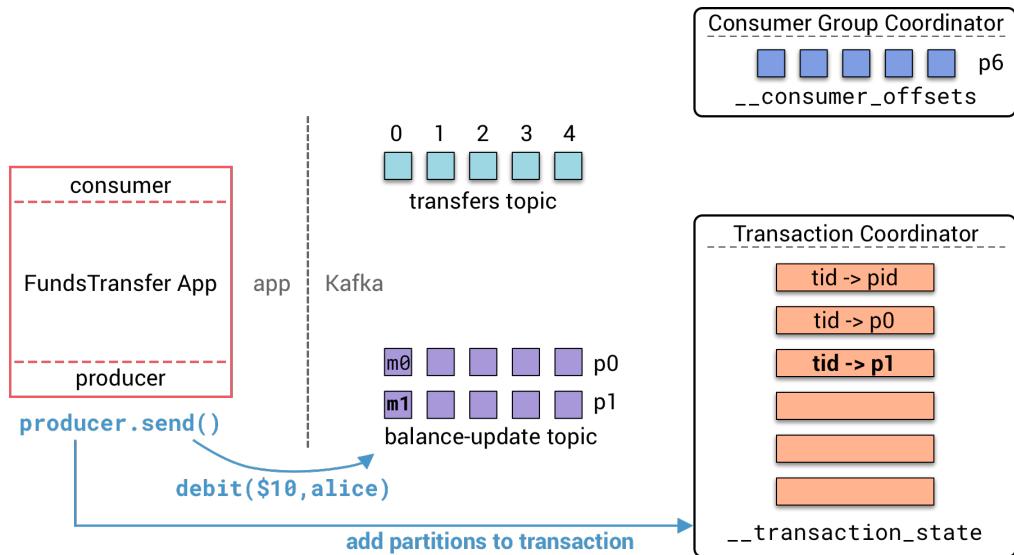


The Producer sends a message to a partition. Here, the message is to credit Bob with \$10.



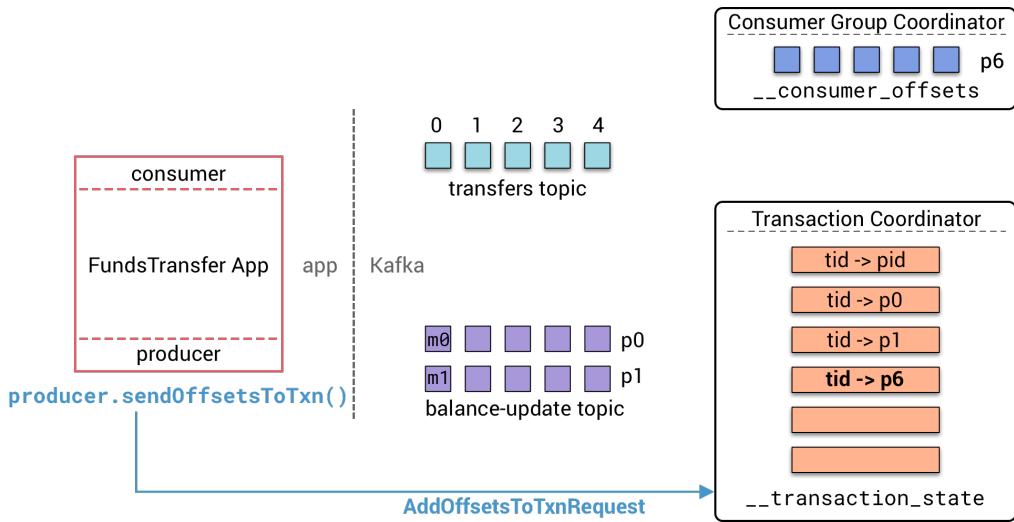
The first time a new TopicPartition is written to as part of a transaction, the producer sends a "Register Partitions" request to the transaction coordinator and this TopicPartition is logged. The transaction coordinator needs this information so that it can write the commit or abort markers to each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



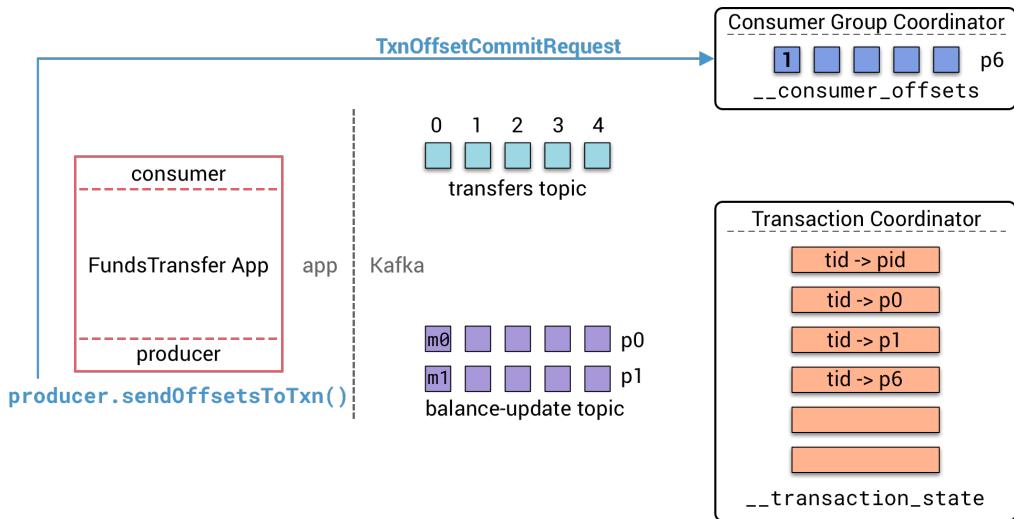
The `sendOffsetsToTxn()` method sends the consumer's offset and consumer group information to the transaction coordinator via an `AddOffsetCommitsToTxnRequest`. This makes the consumer's offset become a part of the transaction. If the transaction fails, the consumer's offset doesn't move forward and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

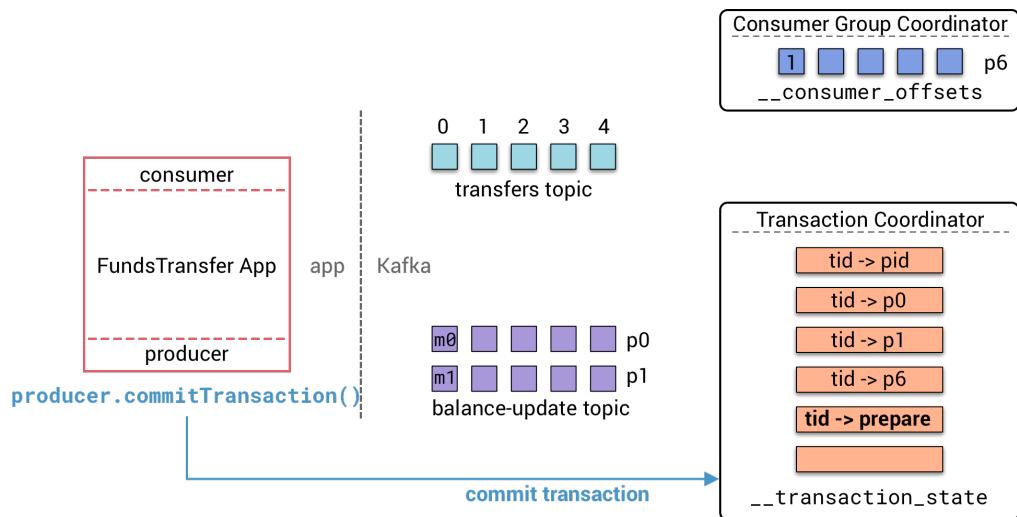
## Transactions - Commit Consumer Offset (10/14)



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

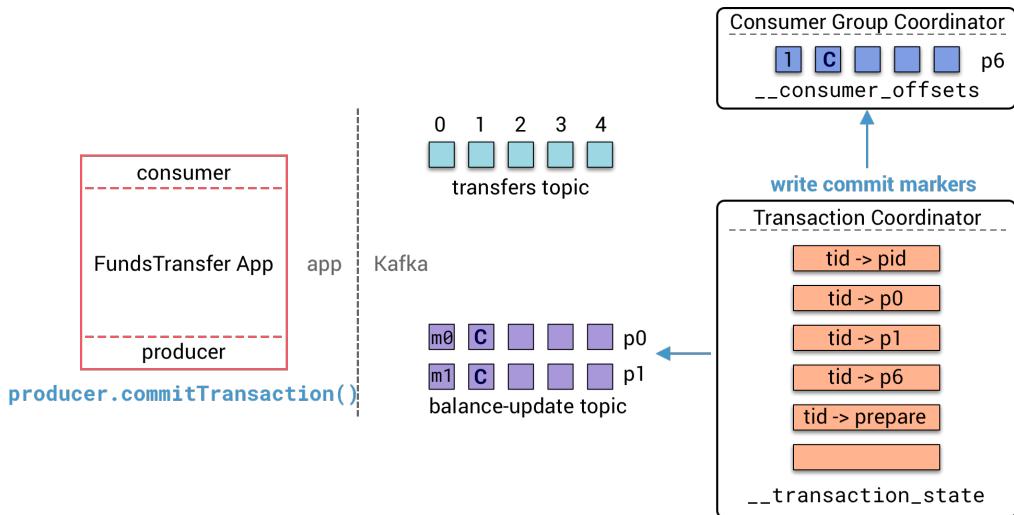
This guarantees the offsets and the output records will be committed as an atomic unit.

# Transactions - Prepare Commit (11/14)



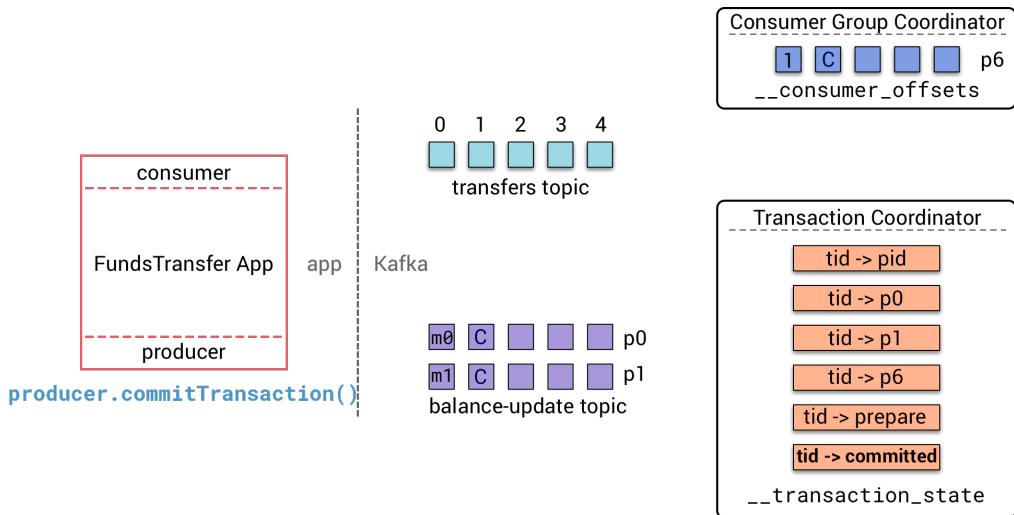
Producer commits the transaction. The transaction coordinator marks the transaction as in status of "preparing."

## Transactions - Write Commit Markers (12/14)



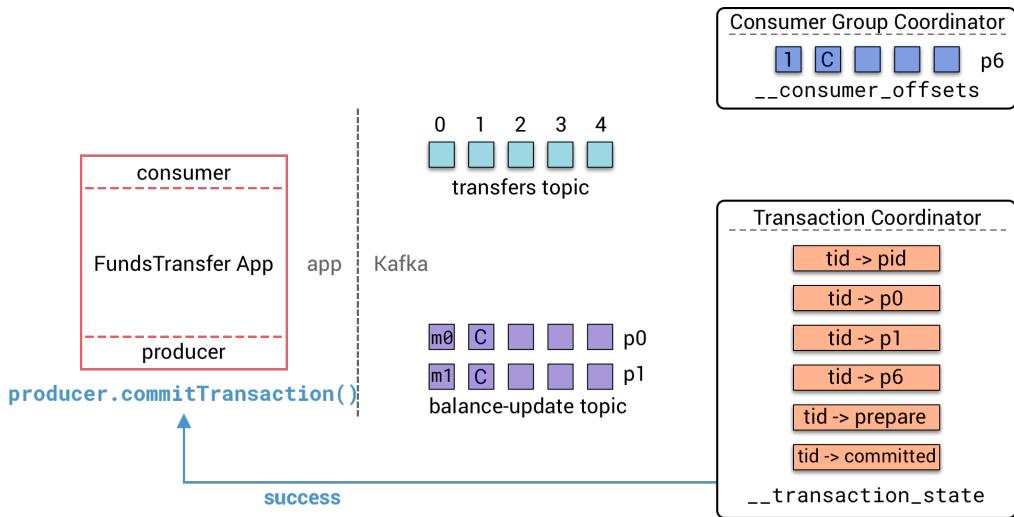
The Transaction Coordinator writes commit markers to the Partitions the Producer writes to as well as to the `__consumer_offsets` Partition. Commit markers are special messages which log the producer id and the result of the transaction (committed or aborted). These messages are internal only and are 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 acknowledgment to the producer.

# Appendix B: How Can You Monitor Replication?

## Description

Monitoring considerations for replication.

hitesh@datacouch.io

# Monitoring Leader Election Rate

- Leader election rate (JMX metric)

```
kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs
```

hitesh@datacouch.io

# 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`
- 

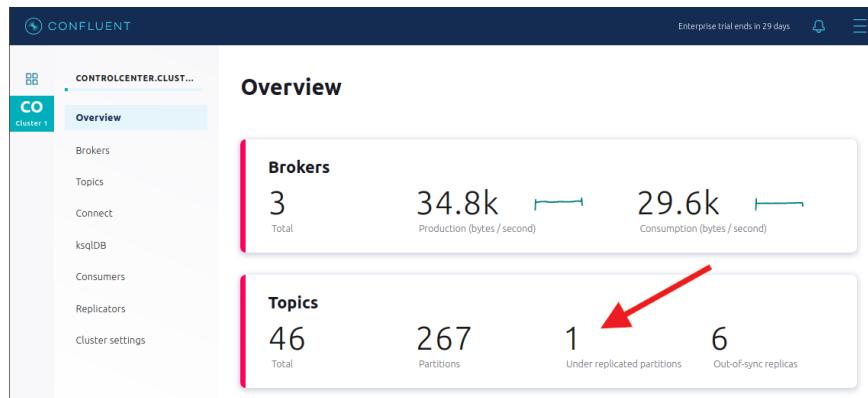
These metrics are the primary indicators of a problem within the cluster.

The `UnderReplicatedPartitions` metric 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, the ISR for some 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.

# Monitoring Offline Partitions

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

---

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.

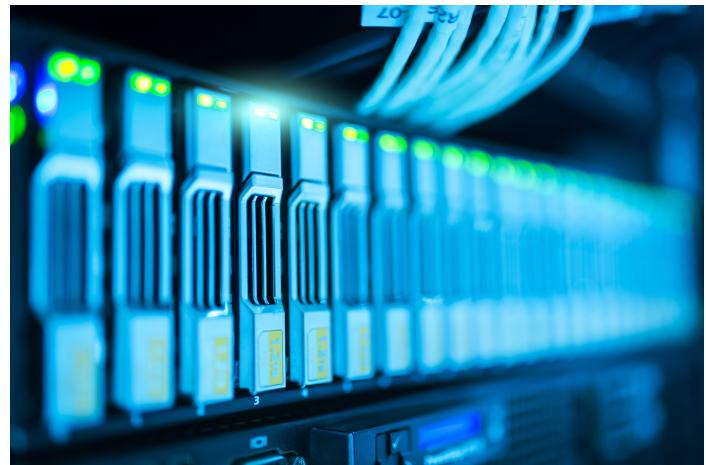
- If `retries == 0`
  - The callback will receive an exception with the following message "org.apache.kafka.common.errors.NetworkException: The server disconnected before a response was received."
  - Message will not be written to Kafka.
- If `retries > 0`
  - No callback but log message will be written "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.

# Appendix C: Multi-Region Clusters

hitesh@datacouch.io

# Multiple Data Centers

- Kafka only:
  - Stretched (a.k.a. multi-AZ)
- Confluent Replicator:
  - Cluster aggregation
  - Active/Passive
  - Active/Active



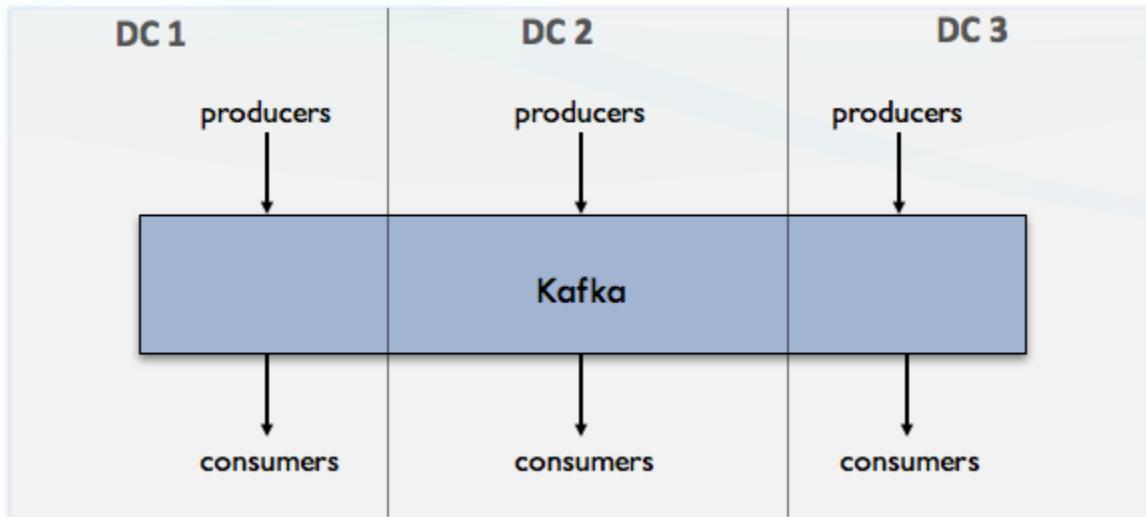
---

There are many use cases for multiple datacenter deployments of Kafka clusters. Using multiple DCs, 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 on [this white-paper](#)

Refer to [this site](#) for a discussion of data aggregation.

# 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.

# 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 [this documentation](#).

**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.

hitesh@datacouch.io

# 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"

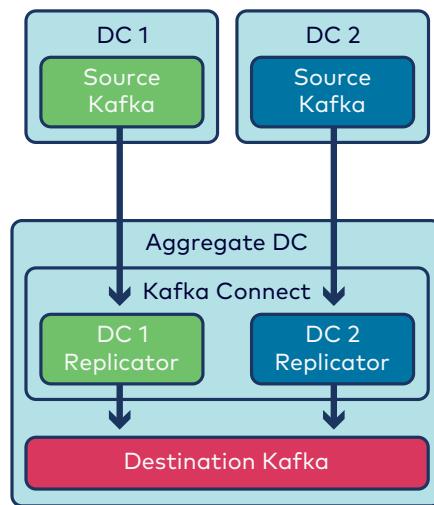
    "key.converter": "io.confluent.connect.replicator.util.ByteArrayConverter",
    "value.converter": "io.confluent.connect.replicator.util.ByteArrayConverter",
    "src.kafka.bootstrap.servers": "dc1-kafka-101:9092",
    "dest.kafka.bootstrap.servers": "dc2-kafka-101:9092"
    "producer.compression.type": "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
```

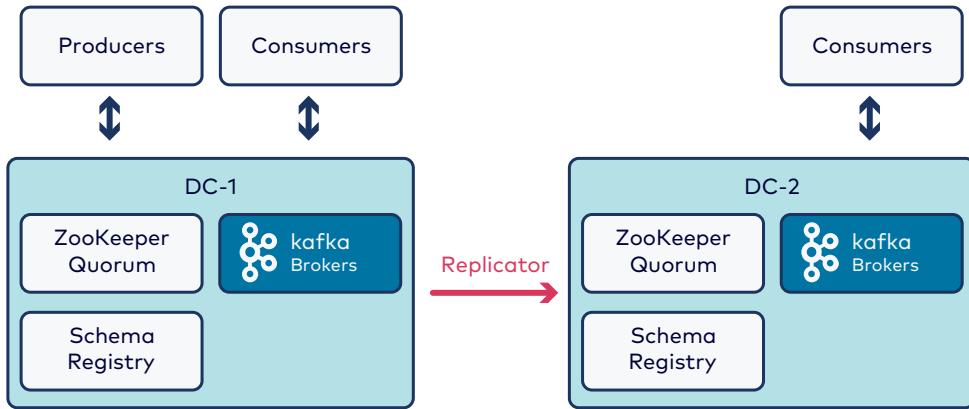
For more information on the replicator, see: [this documentation](#).

# 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.

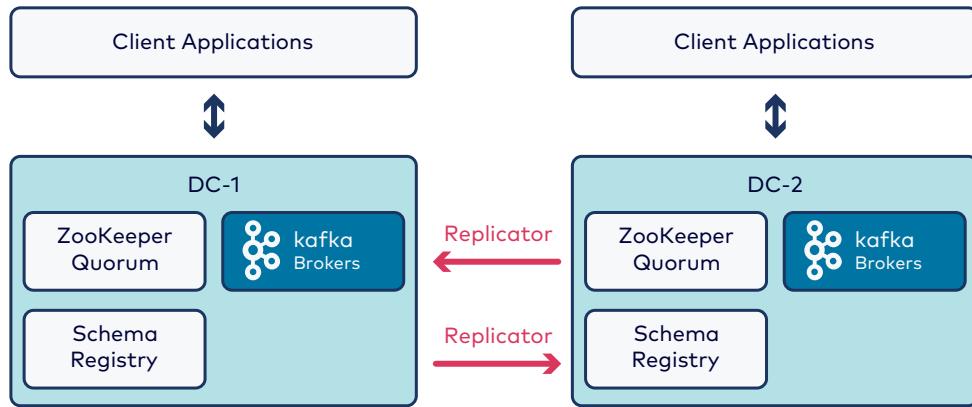
# Active-Passive



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

compacted before replication												
messages	x1 ts0	x2 ts1	A ts2	B ts3	C ts4	D ts5	E ts6	F ts7	G ts8	H ts9	I ts10	J ts11
offsets	1	2	3	4	5	6	7	8	9	10	11	12

Consumer Group	Offset	Timestamp
my-group-1	10	ts9

Data Center 2



messages	A ts2	B ts3	C ts4	D ts5	E ts6	F ts7	G ts8	H ts9	I ts10	J ts11	
offsets	1	2	3	4	5	6	7	8	9	10	

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.

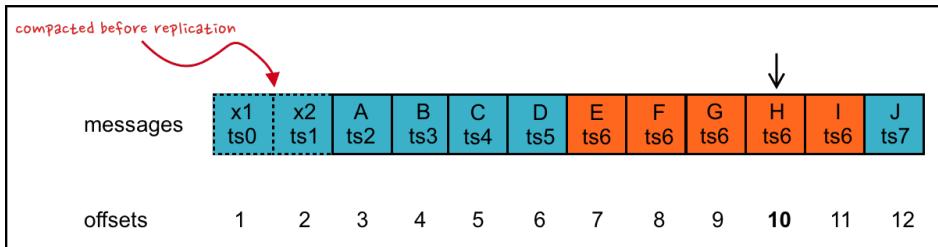
In order for this feature to work, the following requirements must be met in the consumer:

- 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.

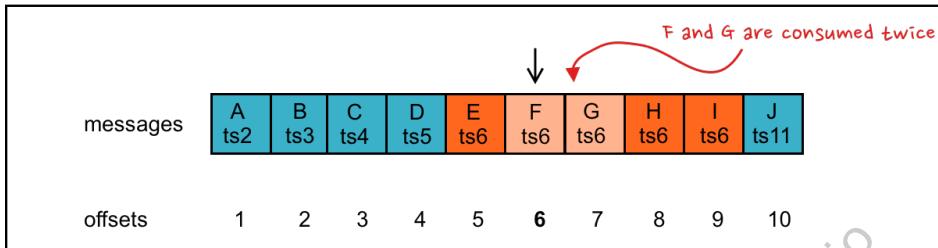
## Client Offset Translation (2)

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
```

hitesh@datacouch.io

# 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
- 

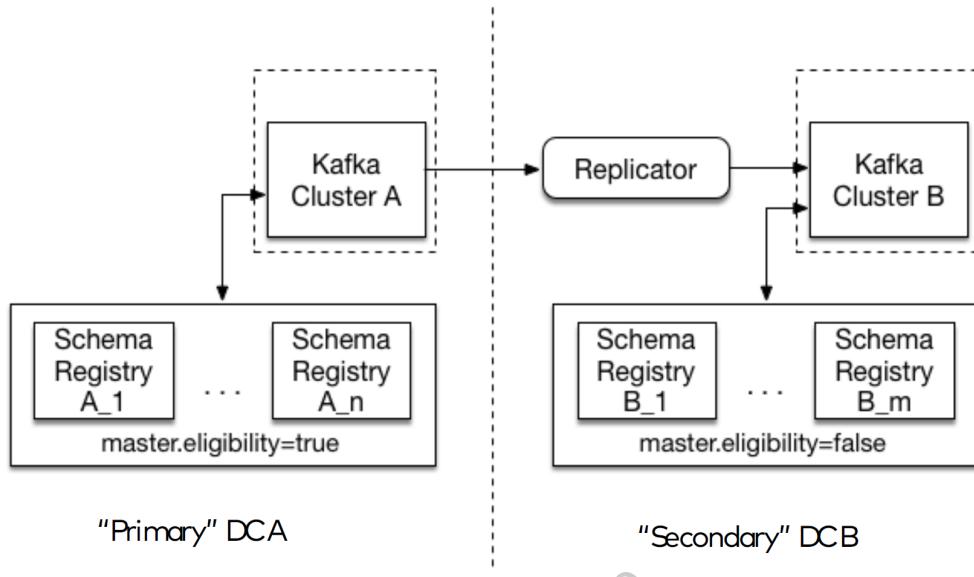
If, for whatever reason, Consumers are not configured with the Replicator interceptor for offset translation, then destination offset will have to be set manually so Consumers at the failover site can recover to an offset relatively close to where they left off before the site failure. The Kafka Client 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 data centers 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 main 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

# Appendix D: SSL and SASL Details

## Overview

This appendix contains two sections (as one appendix):

- SSL
- SASL



This section is a literal copy of old content - no formatting done, just making sure everything renders.

hitesh@datacouch.io

# 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.

# 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.

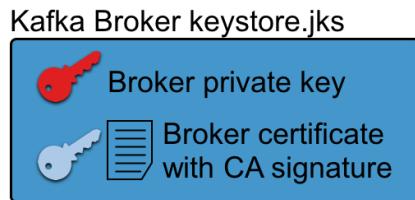
# Data at Rest Encryption

- "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.

# SSL: Keystores and Truststores



- 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 for SSL

1. Generate certificate (X.509) in Broker **keystore**
2. Create your own Certificate Authority (CA) or use a well known CA
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>.

# 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>.

# 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`).

## 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.

hitesh@datacouch.io

## 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>PLAIN is an authentication mechanism, whereas PLAINTEXT is transporting data without encryption. It's possible to use SASL PLAIN with SSL or PLAINTEXT, 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.

hitesh@datacouch.io

# 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";
...
```

hitesh@datacouch.io

## 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?
  - 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.
- Notice again that clear passwords are stored in these files. What would you do to ensure the security of these credentials?
  - 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.
  - 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)

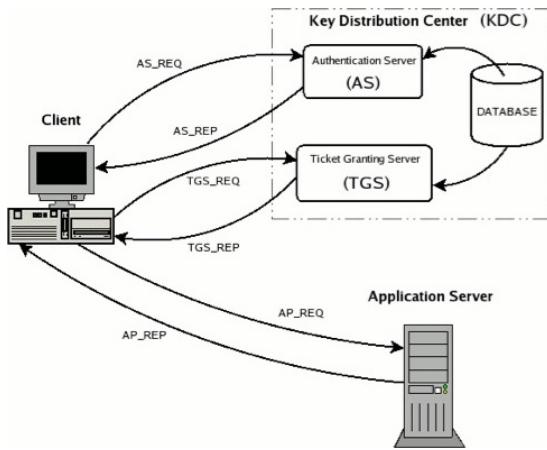
# Why Kerberos?

- 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>.

# 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.



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.

# 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>.

# 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";
```

hitesh@datacouch.io

# 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.

# 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.