



# **Confluent**

## **Custom Training**

### **Prepared for Walmart**

# Introduction

Chapter 1



# Course Contents

---

## >>> 01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Course Objectives

---

- **During this course, you will learn:**
  - The types of data which are appropriate for use with Kafka
  - The components which make up a Kafka cluster
  - Kafka features such as Brokers, Topics, Partitions, and Consumer Groups
  - How to write Producers to send data to Kafka
  - How to write Consumers to read data from Kafka
  - Common patterns for application development
  - How to integrate Kafka with external systems using Kafka Connect
  - Kafka cluster administration and monitoring
  - How to write Kafka Streams applications
- **Throughout the course, Hands-On Exercises will reinforce the topics being discussed**

# Introduction

---

- **About Kafka and Confluent**
- *Class Logistics and Introductions*

# About Kafka

---

- Originally created at LinkedIn in 2010
- Designed to support batch and real-time analytics
- Performs extremely well at very large scale
  - LinkedIn's installation of Kafka processes over 1.4 *trillion* messages per day
- Made open source in 2011, became a top-level Apache project in 2012
- In use at many organizations
  - Twitter, Netflix, Goldman Sachs, Hotels.com, IBM, Spotify, Uber, Square, Cisco...

# About Confluent

---

- **Founded in 2014 by the creators of Kafka**
  - Includes many contributors to the Apache Kafka project
- **Provides support, consulting, training for Kafka and its ecosystem**
- **Develops Confluent Platform**
  - Kafka with additional components
  - Completely free, open source

# Confluent Platform

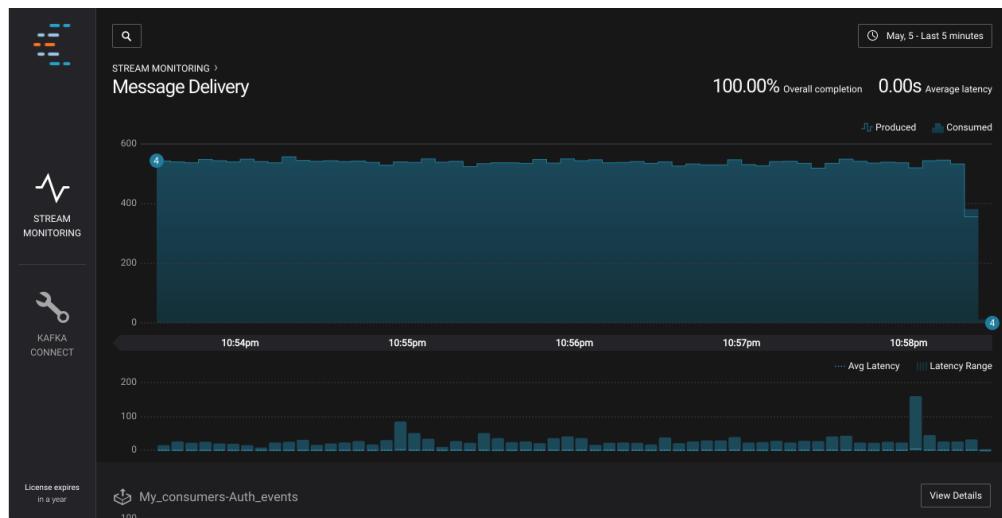
---

- **Confluent Platform: completely free, open source**
- **Easy to install**
  - Available as deb, RPM, Zip archive, tarball
- **Provides Kafka plus extra ecosystem components**
  - Schema Registry
  - REST Proxy
  - Python and C/C++ client libraries
  - Additional Kafka Connect connectors

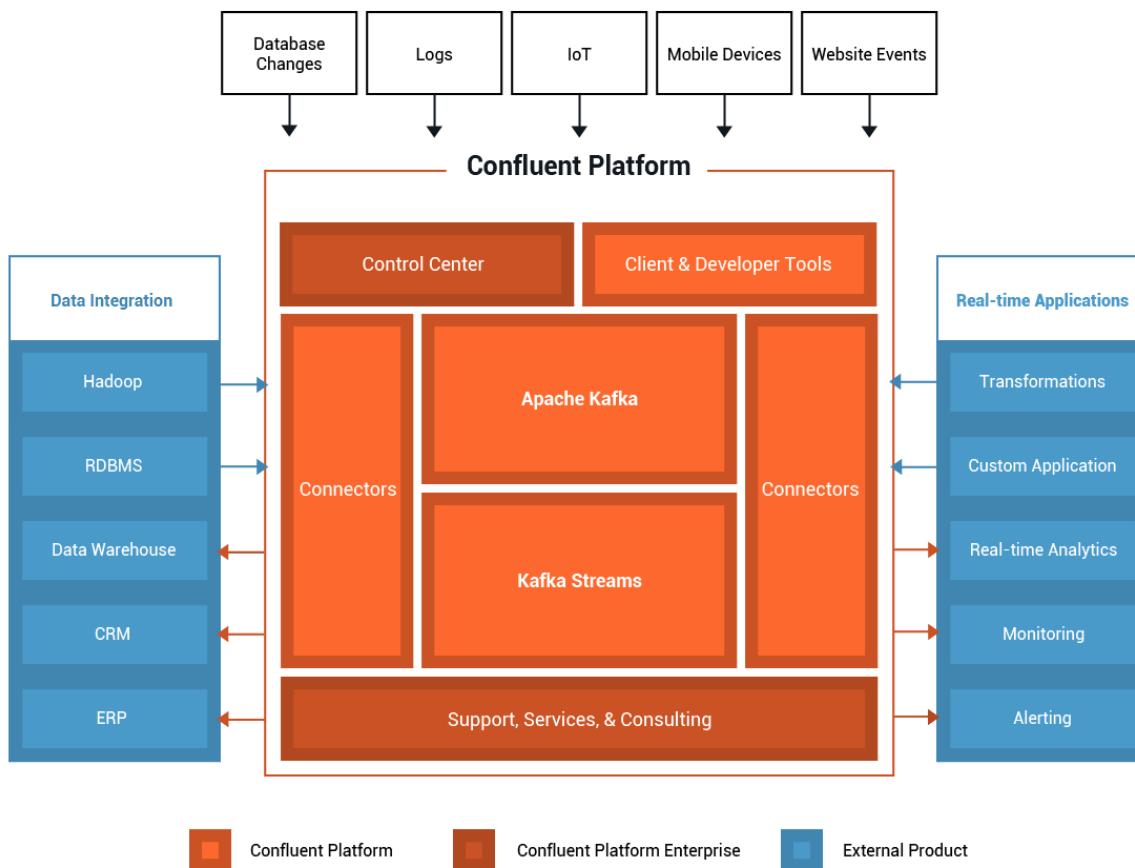
# Confluent Platform Enterprise (1)

## ■ Confluent Platform Enterprise

- Confluent Platform plus world-class enterprise support
- Includes Confluent Control Center
  - Web-based management and monitoring tool
  - Data Stream Monitoring and analytics
  - Kafka Connect Configuration



# Confluent Platform Enterprise (2)



# Introduction

---

- *About Kafka and Confluent*
- **Class Logistics and Introductions**

# Class Logistics

---

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

# Access Course Materials

---

- **Register at Confluent University: <http://confluentuniversity.com>**
  - Create a new account using an email address you can access from this room
  - Click on the link in the verification email to activate your account
  - Register for this class using the <Course ID> and <Registration Key> provided by your instructor
- **Download the Training Guide and Exercise Manual**
  - Do not download the VM at this time
- **Bookmark the web page so that you can complete the survey at the end of class**

# Introductions

---

- **About your instructor**
- **About you**
  - Your name
  - What company do you work for, and what do you do?
  - What experience do you have with Kafka?
  - Have you used any other messaging systems (ActiveMQ, RabbitMQ, etc.)?
  - What programming languages do you use?
  - What are your expectations from the course?

# Kafka Fundamentals

Chapter 2



# Course Contents

---

01: Introduction

**>>> 02: Kafka Fundamentals**

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Kafka Fundamentals

---

- **In this chapter you will learn:**

- How Producers write data to a Kafka cluster
- How data is divided into partitions, and then stored on Brokers
- How Consumers read data from the cluster
- What ZooKeeper is, and how it is used by Kafka clusters

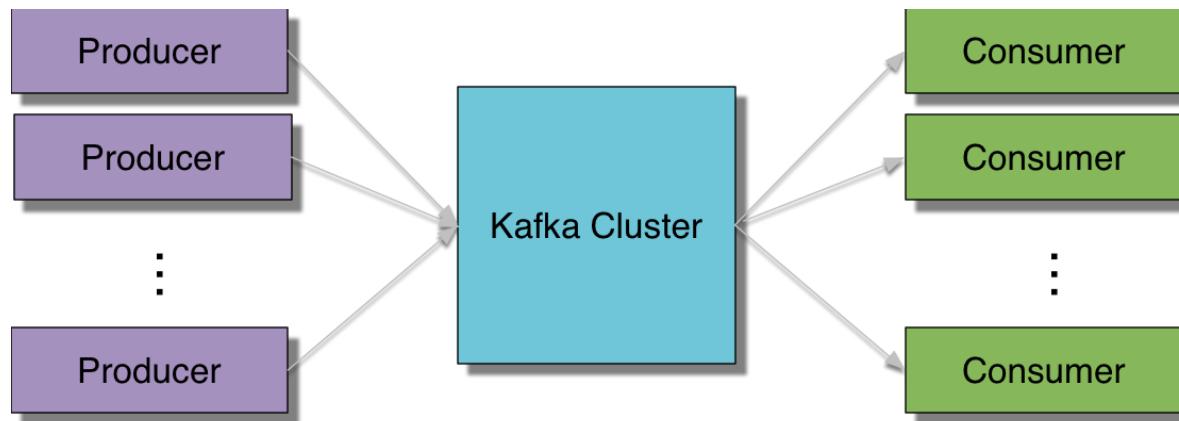
# Kafka Fundamentals

---

- **An Overview of Kafka**
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Kafka Efficiency*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Summary*

# Reprise: A Very High-Level View of Kafka

- Recall: Producers send data to the Kafka cluster
  - Data can then be read by Consumers
- Producers and Consumers never communicate directly with each other



# Messages and Topics

---

- Data is written to Kafka in the form of *messages*
- A message is a key-value pair
  - If no key is required, the Producer can supply a null key
- Each message belongs to a *topic*
  - Topics provide a way to group messages together
- There is no limit to the number of topics that can be used
  - Topics can be created in advance, or created dynamically by Producers (see later)

# Kafka Components

---

- There are four key components in a Kafka system
  - Producers
  - Brokers
  - Consumers
  - ZooKeeper
- We will now investigate each of these in turn

# Kafka Fundamentals

---

- *An Overview of Kafka*
- **Kafka Producers**
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Kafka Efficiency*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Summary*

# Producer Basics

---

- A Producer sends messages to the Kafka cluster
- Producers can be written in any language
  - Native Java, C, and Python clients are supported by Confluent
  - Clients for many other languages exist
  - Confluent develops and supports a REST (REpresentational State Transfer) server which can be used by clients written in any language
- A command-line Producer tool exists to send messages to the cluster
  - Useful for testing, debugging, etc.

# Kafka Messages

---

- A message is the basic unit of data in Kafka
- A message is a key-value pair
- Key and value can be any data type
  - You provide a serializer to turn the key and value into byte arrays
- Key is optional
  - Keys are used to determine which *Partition* (see later) a message will be sent to
  - If no key is specified (or null is passed as the key), the message may be sent to any partition in the topic

# Topics

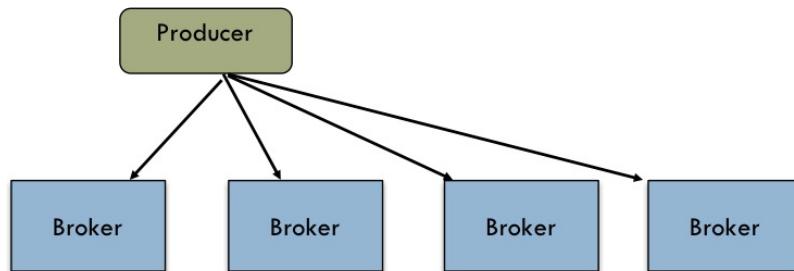
---

- **Each message belongs to a *Topic***
  - Used to segment, or categorize, messages
- **Developers decide which topics exist**
  - No need to create topics in advance
  - By default, a topic is auto-created when it is first used
- **Typically, different systems will write to different topics**

# Topic Partitions Provide Scalability

---

- Topics are split into *Partitions* by Kafka
- Each Partition contains a subset of the Topic's messages
- The Partition to which a message is sent can be specified by the Producer
  - By default, this is based on the hashed value of the key
  - If not specified, messages are sent to Partitions on a round-robin basis



# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- **Kafka Brokers**
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Kafka Efficiency*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Summary*

# Broker Basics

---

- Brokers receive and store messages when they are sent by the Producers
- A Kafka cluster will typically have multiple Brokers
  - Each can handle hundreds of thousands, or millions, of messages per second
- Each Broker manages one or more Partitions

# Brokers Manage Partitions

---

- Any given Partition is handled by a single Broker
  - Typically, a Broker will handle many Partitions
- Each Partition is stored on the Broker's disk as one or more log files
- Each message in the log is identified by its *offset*
  - A monotonically increasing value

# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- **Kafka Consumers**
- *Kafka's Use of ZooKeeper*
- *Kafka Efficiency*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Summary*

# Consumer Basics

---

- **Consumers pull messages from the cluster**
  - Each message is a key-value pair
- **Multiple Consumers can read data from the same topic**
  - By default, each Consumer will receive all the messages in the topic
  - *Consumer Groups* provide scalability (see later)

# The Advantages of a Pull Architecture

---

- Note that Kafka consumers work by pulling messages
  - This is in contrast to some other systems, which use a *push* design
- The advantages of pulling, rather than pushing, data, include:
  - The ability to add more Consumers to the system without reconfiguring the cluster
  - The ability for a Consumer to go offline and return later, resuming from where it left off
  - No problems with the Consumer being overwhelmed by data
    - It can pull, and process, the data at whatever speed it needs to

# Keeping Track of Position

---

- As messages are written to a topic, the Consumer will automatically retrieve them
- The *Consumer Offset* keeps track of the latest message read
- If necessary, the Consumer Offset can be changed
  - For example, to reread messages
- The Consumer Offset is stored in a special Kafka topic
  - (Aside: Previously offsets were stored in ZooKeeper, but as of Kafka 0.9 this is no longer the case)

# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- **Kafka's Use of ZooKeeper**
- *Kafka Efficiency*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Summary*

# What is ZooKeeper?

---

- Apache ZooKeeper is an Apache project
- It is “a centralized service for maintaining configuration information”
  - A distributed, highly reliable system in which configuration information and other data can be stored
- Used by many projects
  - Including Hadoop and Kafka
- Typically consists of three or five servers in a quorum
  - This provides resiliency should a machine fail



# How Kafka Uses ZooKeeper

---

- Kafka Brokers use ZooKeeper for a number of important internal features
  - Leader election, failure detection
- In general, end-user developers should not have to be concerned with ZooKeeper
  - (In earlier versions of Kafka, the Consumer needed access to the ZooKeeper quorum. This is no longer the case)
- Much more detail in *Confluent Administrator Training for Kafka*

# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- **Kafka Efficiency**
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Summary*

# Decoupling Producers and Consumers

---

- A key feature of Kafka is that Producers and Consumers are decoupled
- A slow Consumer will not affect Producers
- More Consumers can be added without affecting Producers
- Failure of a Consumer will not affect the system
- Multiple brokers, multiple topics, and *Consumer Groups* (see later) provide very high scalability

# The Page Cache for High Performance

---

- Unlike some systems, Kafka itself does not require a lot of RAM
- Logs are held on disk, and read when required
- Kafka makes use of the operating system's page cache to hold recently-used data
  - Typically, recently-produced data is the data which Consumers are requesting
- A Kafka Broker running on a system with a reasonable amount of RAM for the OS to use as cache will typically be able to swamp its network connection
  - In other words the network, not Kafka itself, will be the limiting factor on the speed of the system

# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Kafka Efficiency*
- **Hands-On Exercise: Using Kafka's Command-Line Tools**
- *Chapter Summary*

# Hands-On Exercise: Using Kafka's Command-Line Tools

---

- In this Hands-On Exercise you will use Kafka's command-line tools to Produce and Consume data
- Please refer to the Hands-On Exercise Manual

# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Kafka Efficiency*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- **Chapter Summary**

# Chapter Summary

---

- A Kafka system is made up of Producers, Consumers, and Brokers
  - ZooKeeper provides co-ordination services for the Brokers
- Producers write messages to topics
  - Topics are broken down into partitions for scalability
- Consumers read data from one or more topics

# Kafka's Architecture

Chapter 3



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

**>>> 03: Kafka's Architecture**

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Kafka's Architecture

---

- How Kafka's log files are stored on the Kafka Brokers
- How Kafka uses replicas for reliability
- What the read path and write path look like
- How Consumer Groups and Partitions provide scalability

# Kafka's Architecture

---

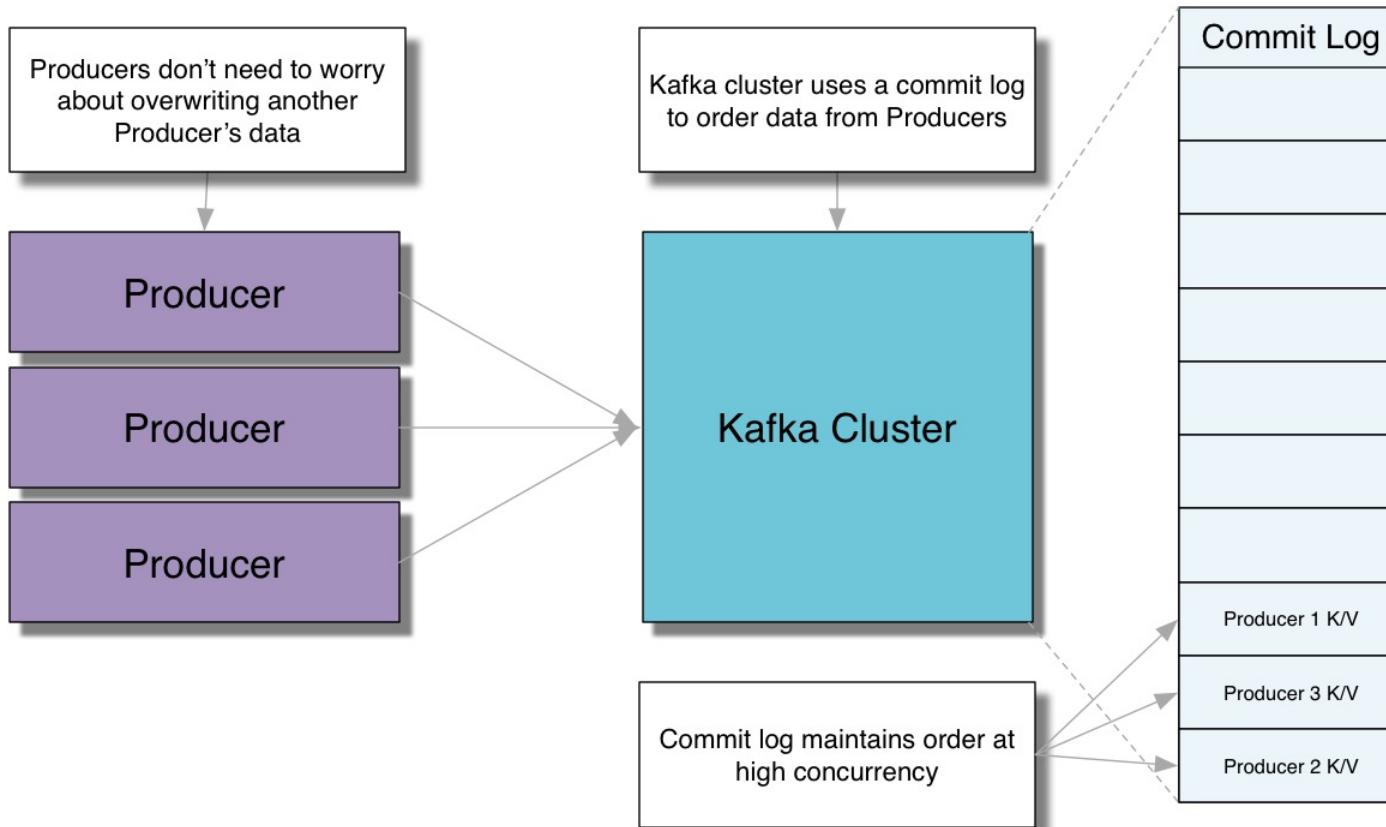
- **Kafka's Log Files**
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Summary*

# What is a Commit Log?

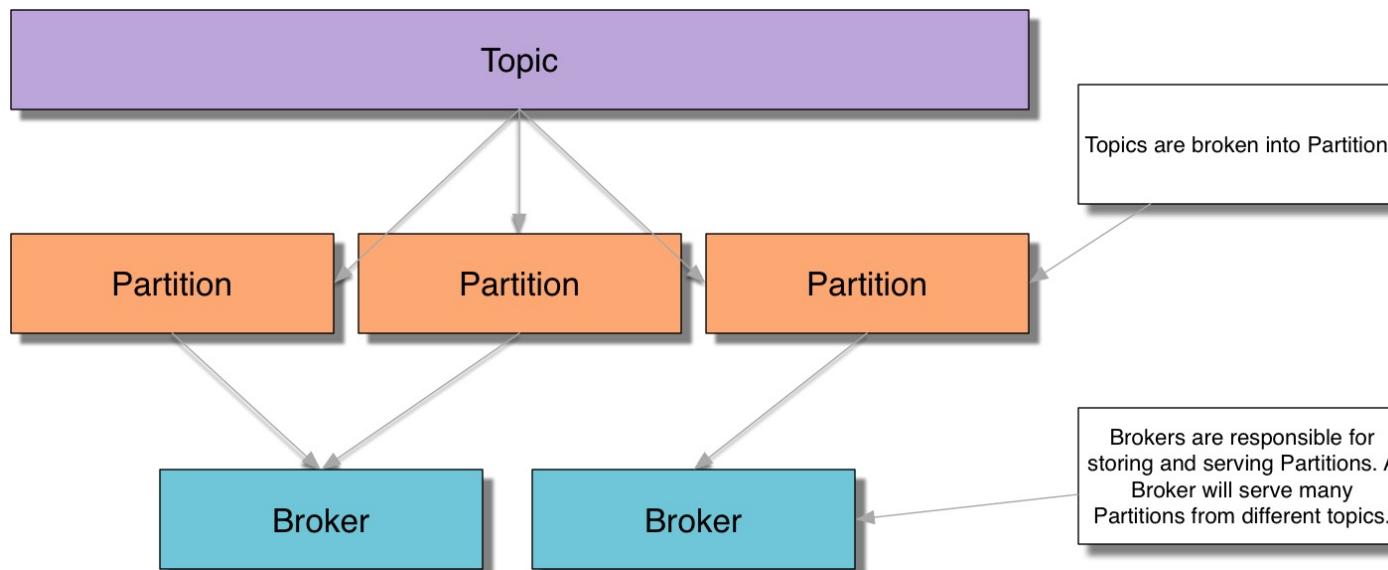
---

- A Commit Log is a way to keep track of changes as they happen
- Commonly used by databases to keep track of all changes to tables
- Kafka uses commit logs to keep track of all messages in a particular topic
  - Consumers can retrieve previous data by backtracking through the commit log

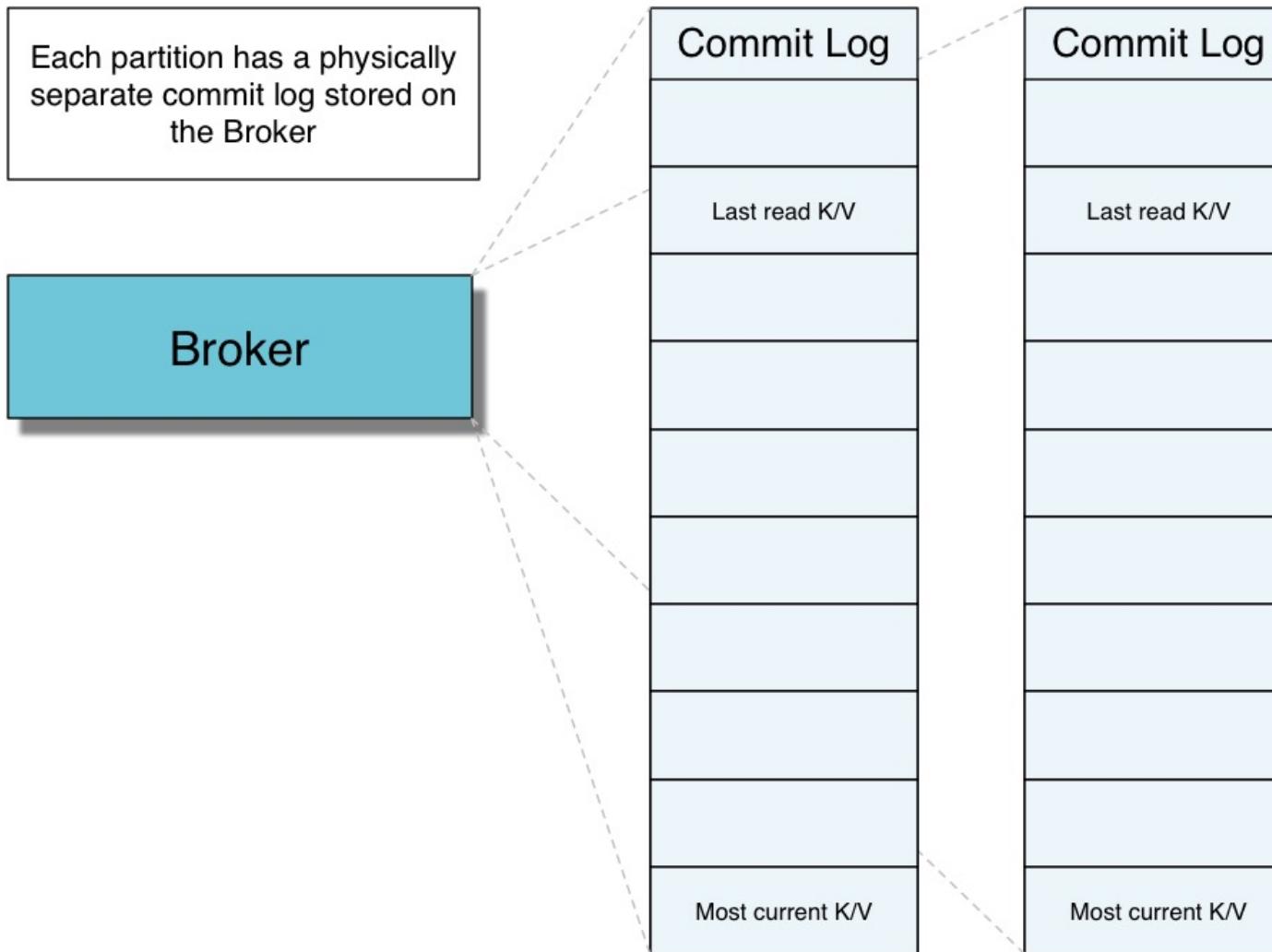
# The Commit Log for High Concurrency



# Brokers Store Partitions of Topics



# Partitions Are Stored as Separate Logs



# Kafka's Architecture

---

- *Kafka's Log Files*
- **Partitions and Consumer Groups for Scalability**
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Summary*

# Scaling using Partitions

---

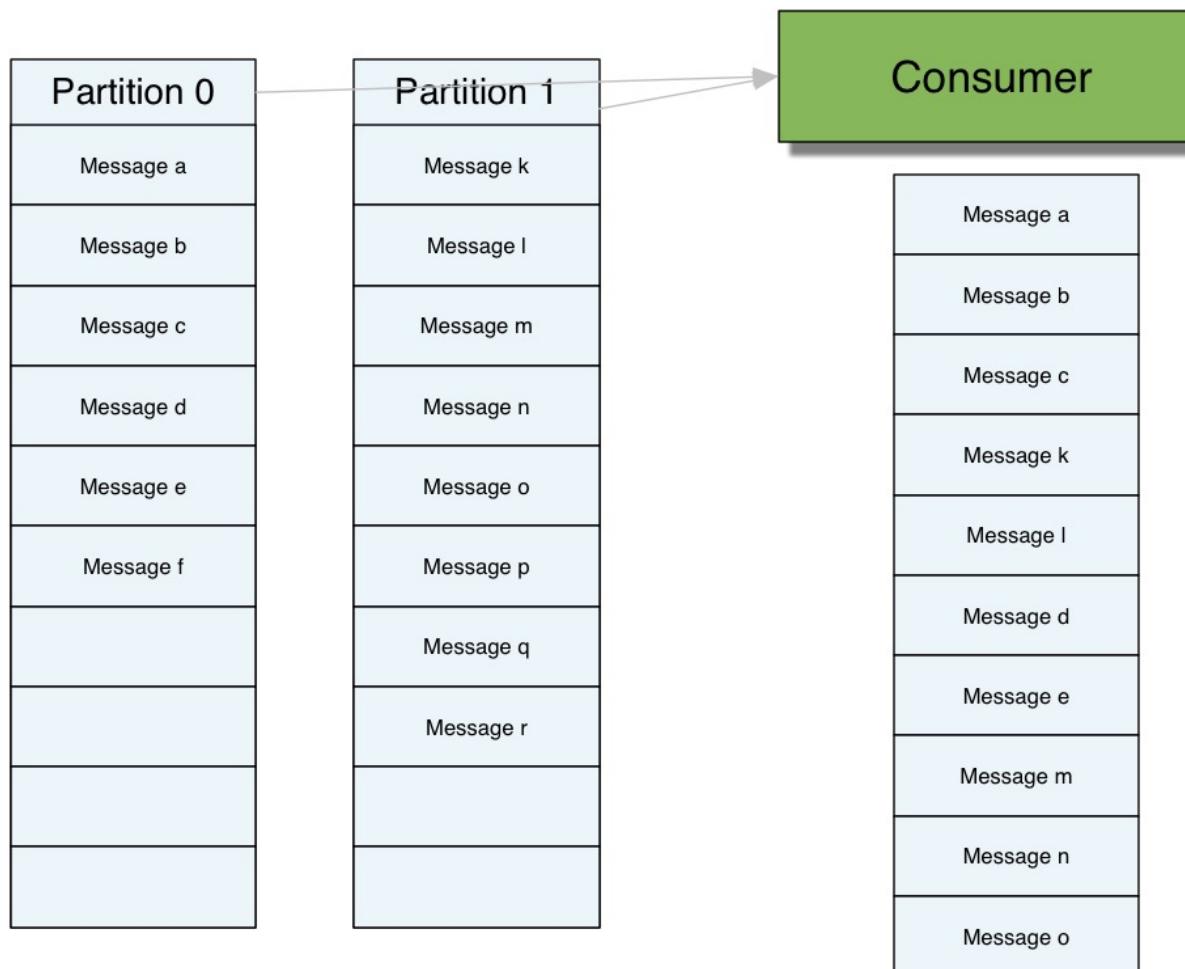
- **Recall: All Consumers read from the Leader of a Partition**
  - No clients write to, or read from, Followers
- **This can lead to congestion on a Broker if there are many Consumers**
- **Splitting a topic into multiple Partitions can help to improve performance**
  - Leaders for different Partitions can be on different Brokers

# An Important Note About Ordering (1)

---

- Data within a Partition will be stored in the order in which it is written
- Therefore, data read from a Partition will be read in order *for that partition*
- If there are multiple Partitions, you will not get total ordering when reading data

# An Important Note About Ordering (2)

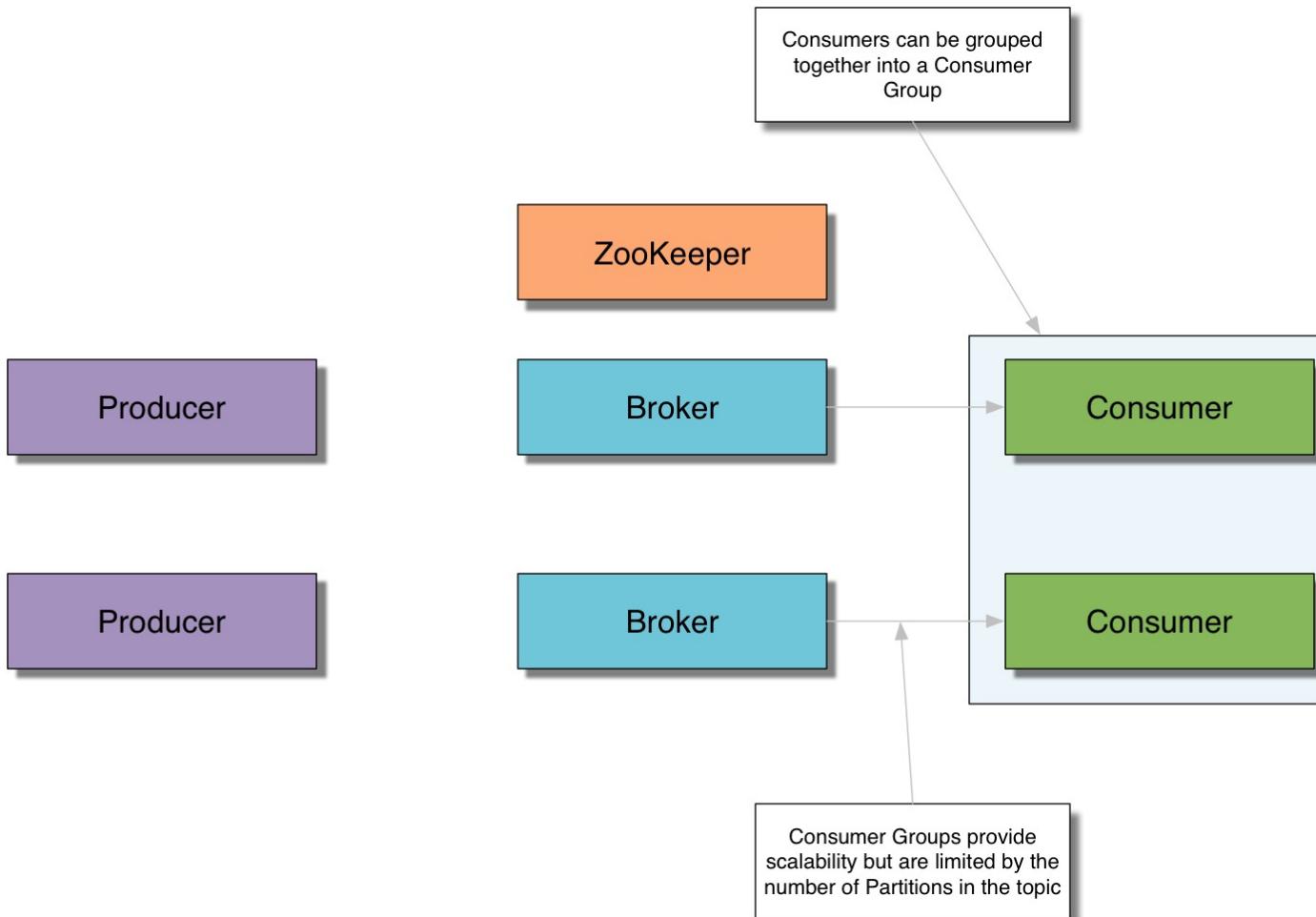


# Consumer Groups (1)

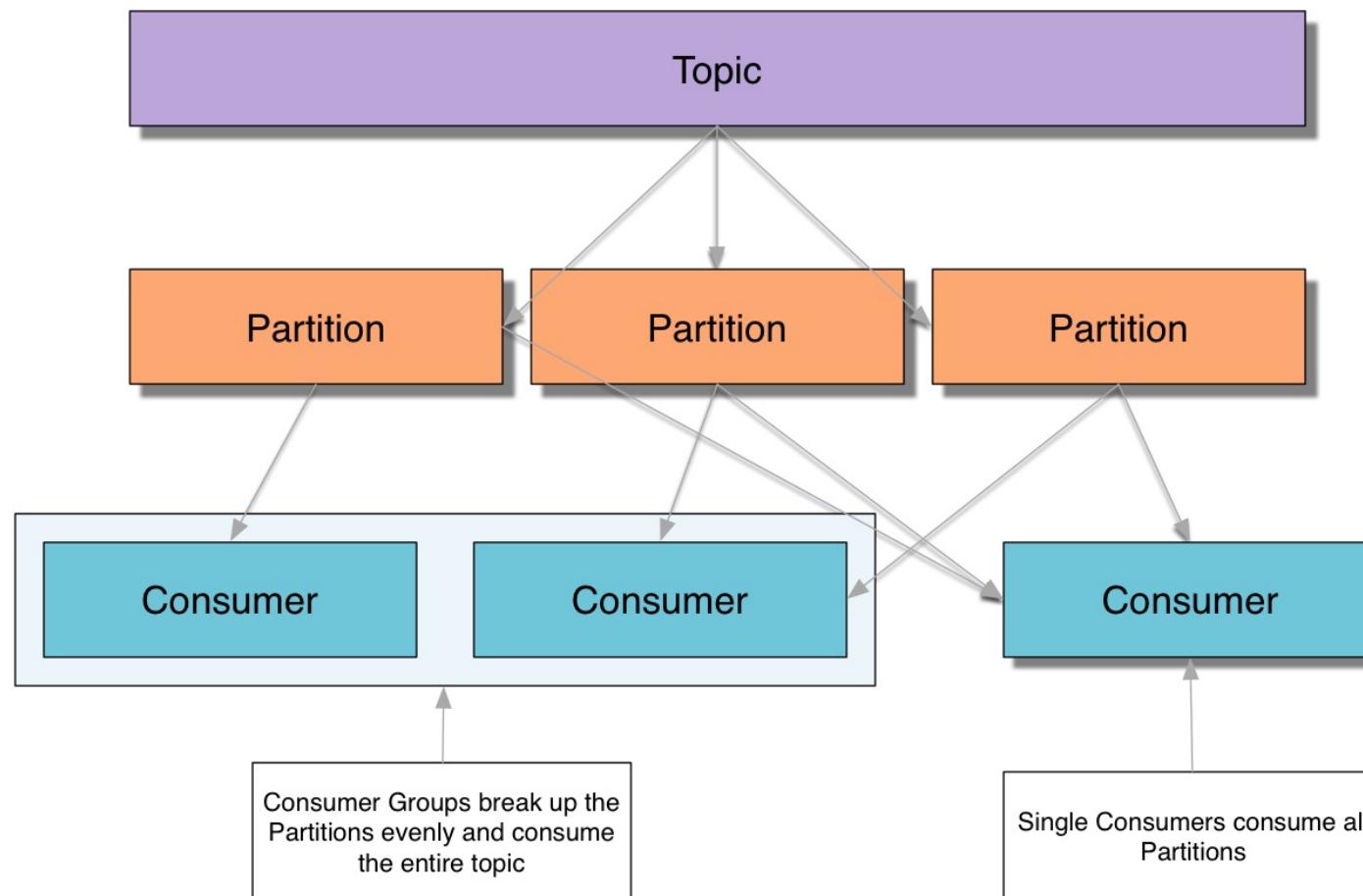
---

- **Multiple Consumers can work together as a single *Consumer Group***
  - Consumers in the group are typically on separate machines
- **The group.id property is identical across all Consumers in the group**
- **Each Consumer will read from one or more Partitions for a given topic**
- **Data from a Partition will go to a single Consumer in the group**
  - i.e., you are guaranteed that messages with the same key will go to the same Consumer
  - Unless you change the number of partitions (see later)

# Consumer Groups (2)



# Consumer Groups (3)



# Consumer Groups: Limitations

---

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

# Consumer Groups: Caution When Changing Partitions

---

- Recall: All messages with the same key will go to the same Consumer
  - However, if you change the number of Partitions in the topic, this may not be the case
    - 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

# Kafka's Architecture

---

- *Kafka's Log Files*
- *Partitions and Consumer Groups for Scalability*
- **Hands-On Exercise: Consuming from Multiple Partitions**
- *Chapter Summary*

# Hands-On Exercise: Consuming from Multiple Partitions

---

- In this Hands-On Exercise, you will create a topic with multiple Partitions, write data to the topic, then read the data back to see how ordering of the data is affected
- Please refer to the Hands-On Exercise Manual

# Kafka's Architecture

---

- *Kafka's Log Files*
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- **Chapter Summary**

# Chapter Summary

---

- **Kafka uses commit logs to store all its data**
  - These allow the data to be read back by any number of Consumers
- **Topics can be split into Partitions for scalability**
- **Partitions are replicated for reliability**
- **Consumers can be collected together in Consumer Groups**
  - Data from a specific Partition will go to a single Consumer in the Consumer Group
- **If there are more Consumers in a Consumer Group than there are Partitions in a topic, some Consumers will receive no data**

# Intra-Cluster Replication

Chapter 4



# Course Contents

---

- 01: Introduction
- 02: Kafka Fundamentals
- 03: Kafka's Architecture
- >>> 04: Intra-Cluster Replication**
- 05: Log Retention and Compaction
- 06: Developing With Kafka
- 07: More Advanced Kafka Development
- 08: Schema Management In Kafka
- 09: Kafka Connect for Data Movement
- 10: Basic Kafka Administration
- 11: Runtime Configurations
- 12: Monitoring and Alerting
- 13: Kafka Security
- 14: Kafka Streams

# Intra-Cluster Replication

---

- In this chapter you will learn:
  - How replication works
  - Important replication concepts
    - Leader, Follower, ISR, Controller
  - How to configure logs and JMX

# Intra-Cluster Replication

---

- Basic Replication Concepts
- *Replica Recovery and Failure Detection*
- *Chapter Review*

# Why Replication?

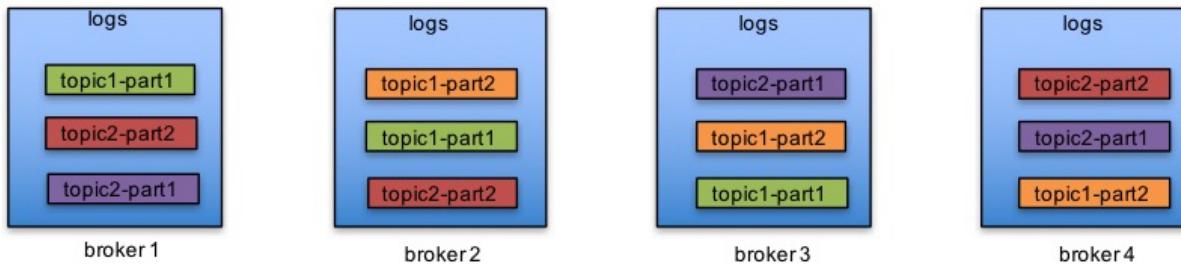
---

- **The Broker can go down**
  - Controlled: rolling restart for code or configuration changes
  - Uncontrolled: Isolated broker failure, machine crash
- **Without replication, if the broker is down...**
  - Some partitions will be unavailable
  - Permanent data loss could occur
- **Replication provides higher availability and durability for the cluster**

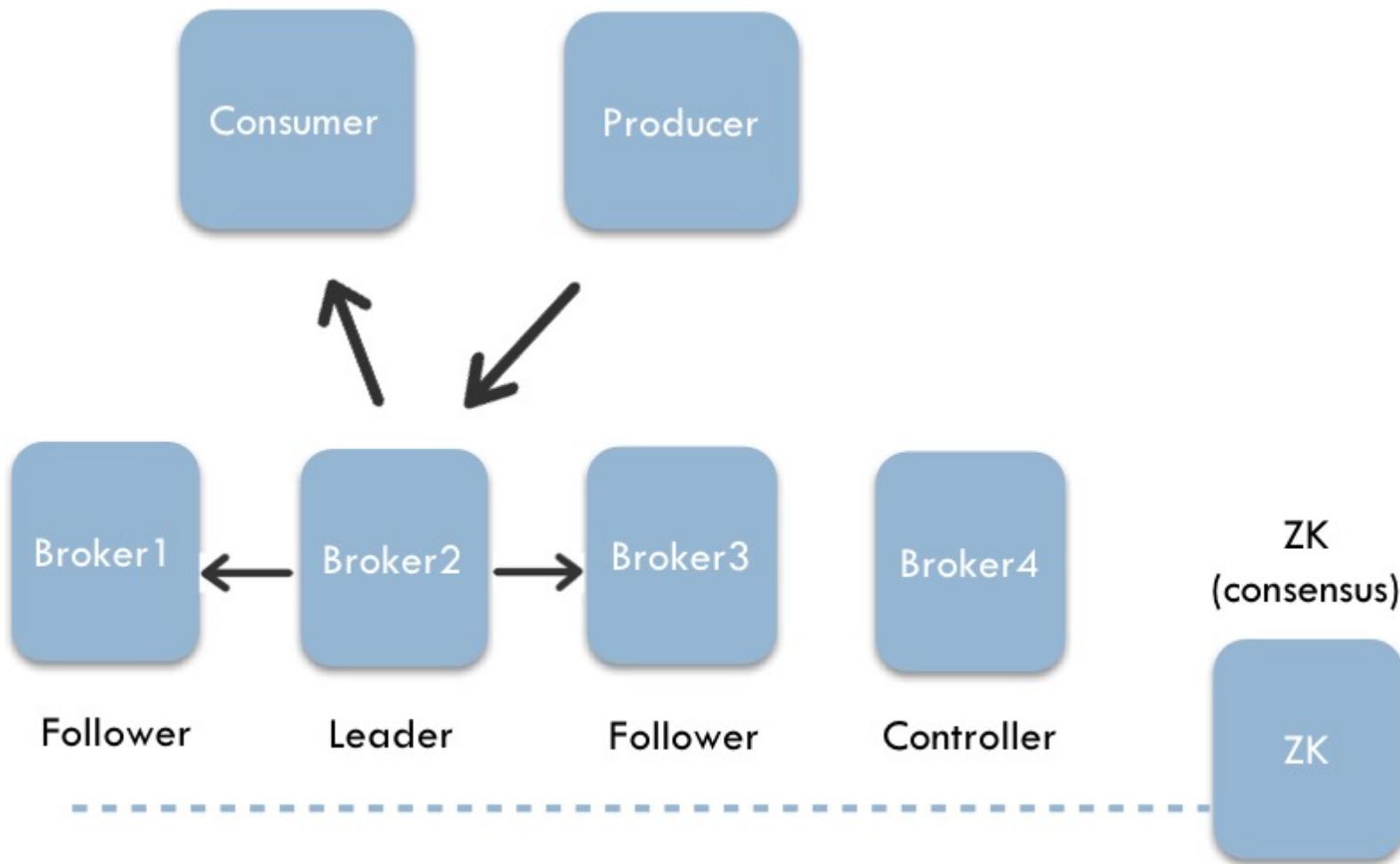
# Replicas and Layout

---

- Each Partition has Replicas
- The Replicas are spread evenly among Brokers



# Roles (1)

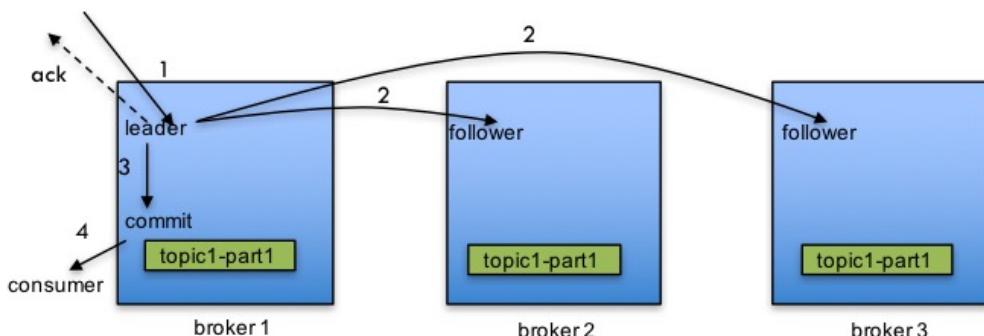


# Roles (2)

---

- **Leader**
  - Accepts all writes and reads for a specific partition, manages In Sync Replicas (ISRs)
- **Replicas/Followers**
  - Provide fault tolerance
  - Attempt to keep up with the Leader
- **Controller**
  - Selects a new leader and updates the ISR list
  - Persists the new Leader and ISR list to ZooKeeper
  - Sends new Leader and ISR changes to all brokers
- **ZooKeeper**
  - Provides consensus

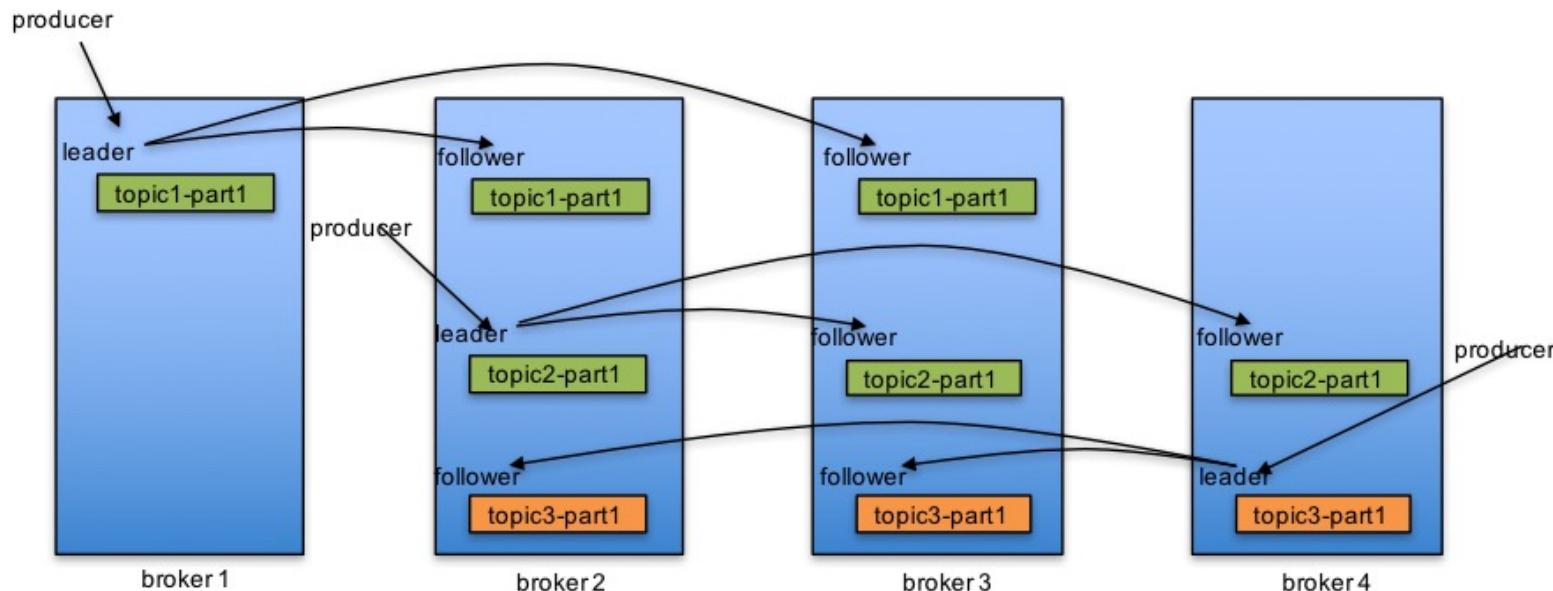
# Data Flow in Replication



When Producer receives ack	Latency	Durability on failures
No ack	No network delay	Some data loss
Wait for Leader	1 network roundtrip	A little data loss
Wait for committed	2 network roundtrips	No data loss

- Only committed messages are exposed to Consumers
  - Independent of the ack type chosen by the Producer

# Extend to Multiple Partitions



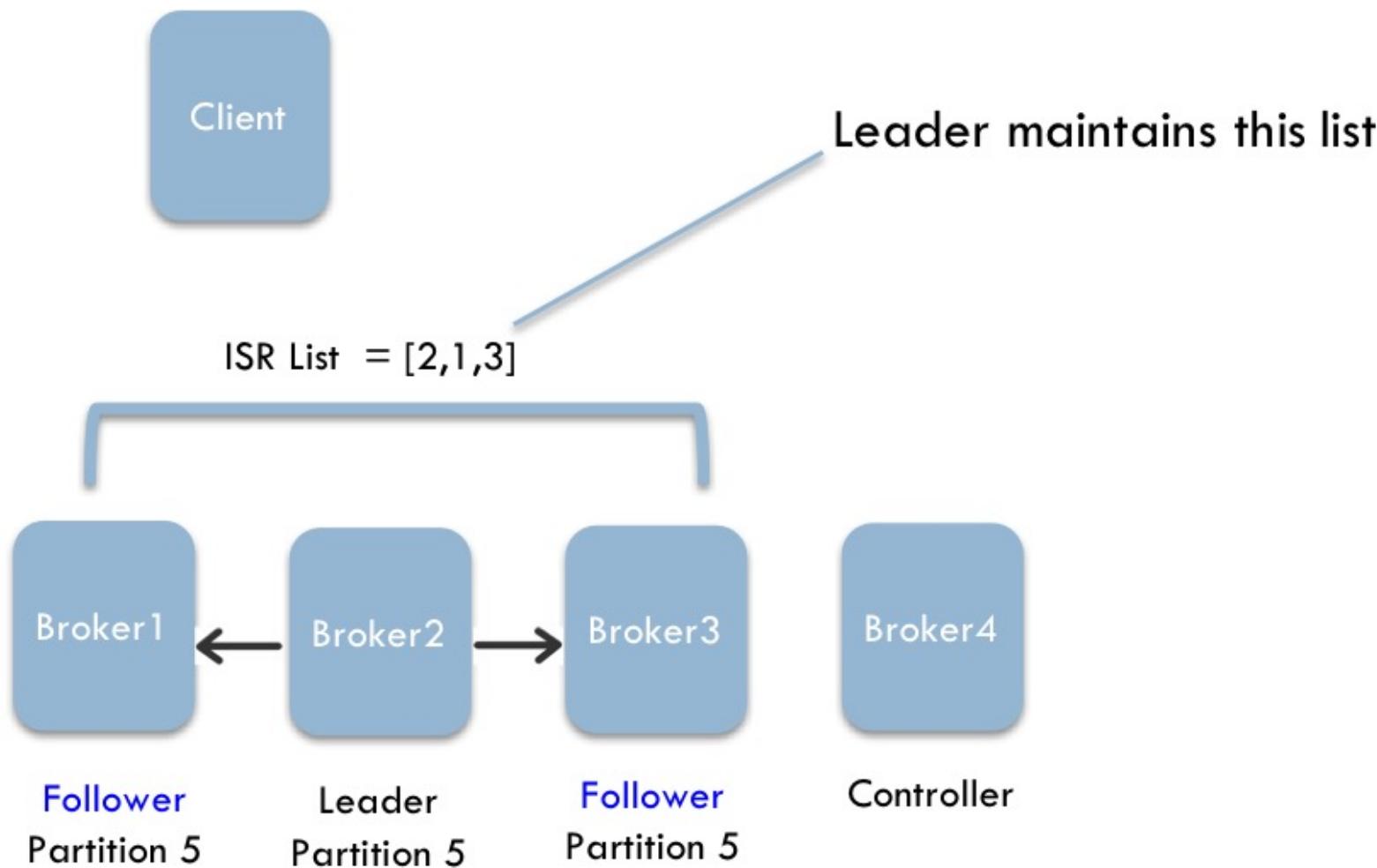
- Leaders are evenly spread among the Brokers

# Strongly Consistent Replicas

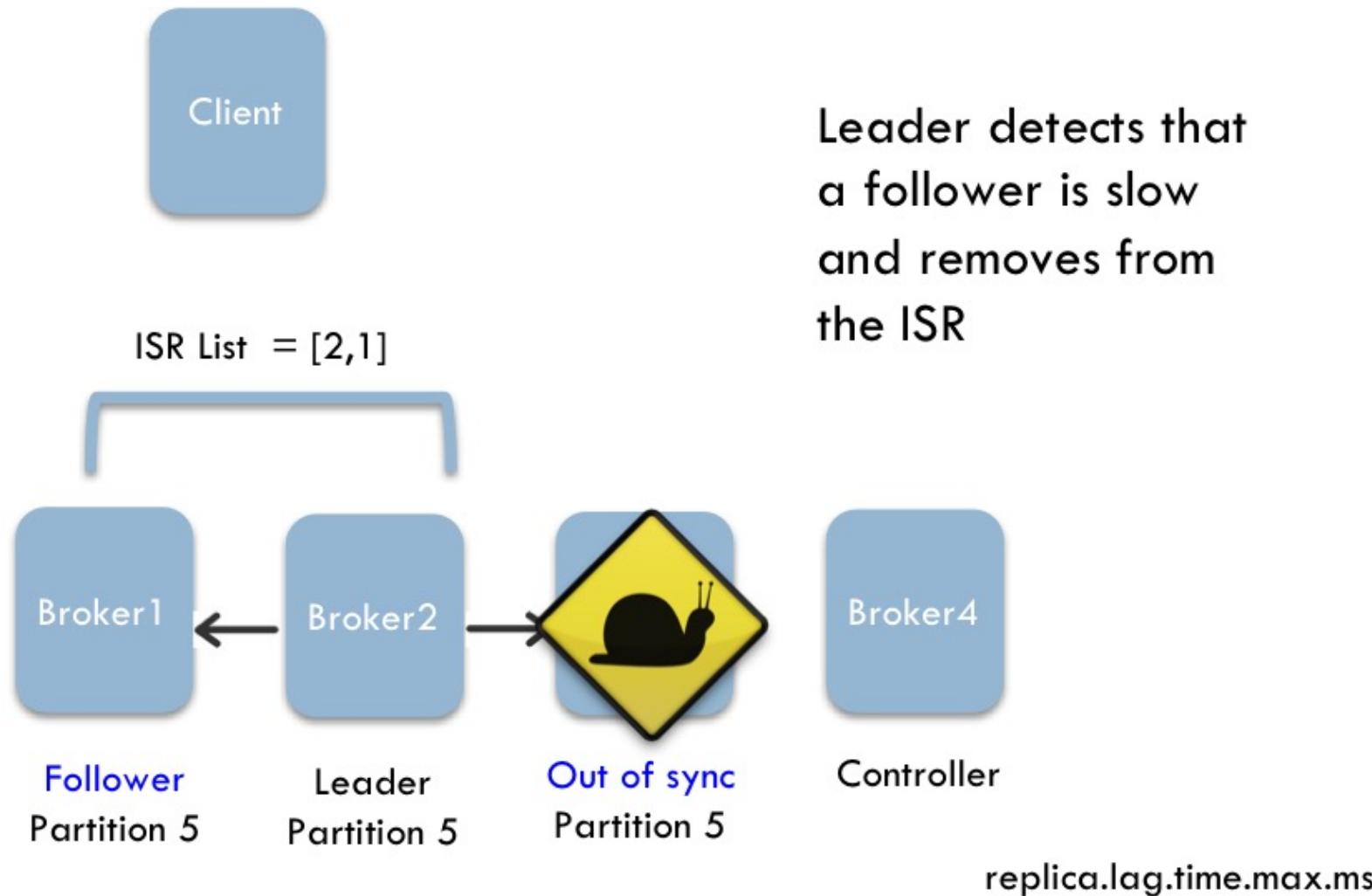
---

- One replica is the Leader
- All writes go to the Leader
- The Leader propagates writes to Followers in order
- The Leader decides when to commit a message

# In Sync Replicas



# In Sync Replicas: Slow Broker

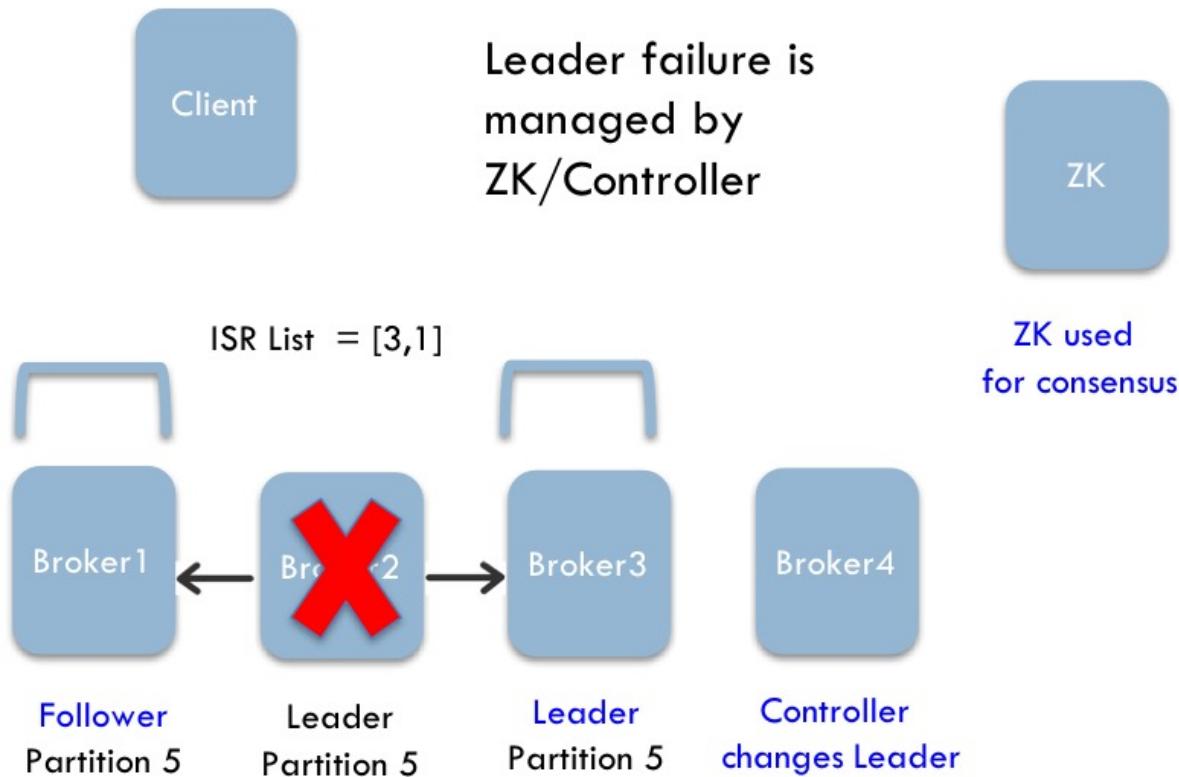


# Replicas Tracked by Lag → ISR

---

- Slow Replicas will be removed from the ISR
- `replica.lag.time.max.ms`
  - Controls the lag between Leader and Replica

# In Sync Replicas: Leader Failure



# How Many Brokers Can We Lose?

---

- **min.insync.replicas setting**
  - Will prevent acknowledgment if ISR list falls below this threshold
  - When the Replica count drops below the minimum ISR, writes will block

# Replica Flow Summary

---

- **Leader maintains a list of ISRs**
  - Initially, all Replicas are in ISR
  - A message is committed if it is received by every Replica in ISR
- **If a Follower fails:**
  - It is dropped from ISR
  - Leader commits using the new ISRs
- **If a Leader fails:**
  - A new Leader is chosen from the live replicas in ISR
- **Tradeoff:**
  - $n$  Replicas means we can tolerate  $n-1$  failures
  - Long latency
    - Typically not a big issue within a data center

# Intra-Cluster Replication

---

- *Basic Replication Concepts*
- **Replica Recovery and Failure Detection**
- *Chapter Review*

# What Does Committed Really Mean?

---

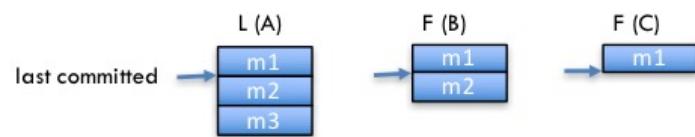
- Committed: Data is received by all the replicas in the ISR
- This is not related to the ack setting chosen by the Producers
- Committed state is checkpointed to disk
- Data cannot be seen (fetched) until it is committed

# Handling Replica Restarts

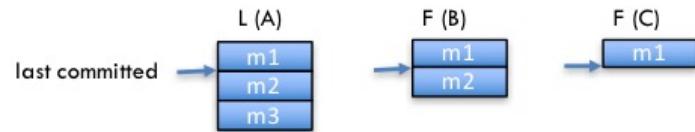
---

- **Leader maintains the last committed offset**
  - This is propagated to Followers
  - Also checkpointed to disk
- **Replica joins as a Follower on restart**
  - Truncates its log to the last committed position
  - Fetches data from the leader
  - Added to the ISR when it is fully caught up

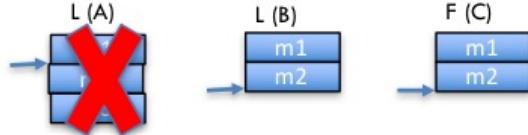
# Example of Replica Recovery (1)



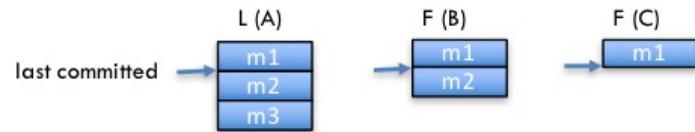
# Example of Replica Recovery (2)



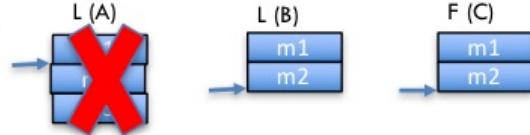
2. A fails and B is new leader; ISR = {B,C}; B commits m2, but not m3. C syncs with new leader B.



# Example of Replica Recovery (3)

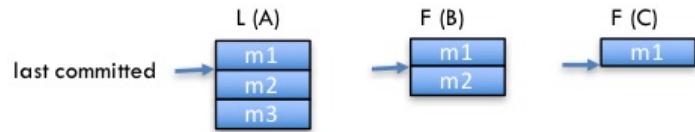


2. A fails and B is new leader; ISR = {B,C}; B commits m2, but not m3. C syncs with new leader B.

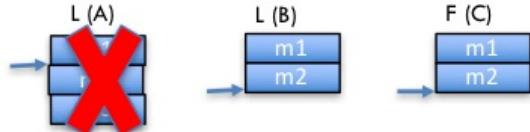


NB: if C had been chosen as leader m2 could also be lost.

# Example of Replica Recovery (4)

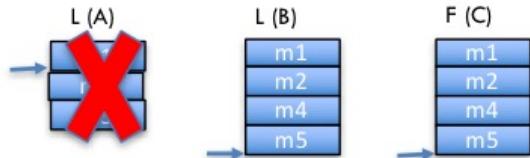


2. A fails and B is new leader; ISR = {B,C}; B commits m2, but not m3. C syncs with new leader B.

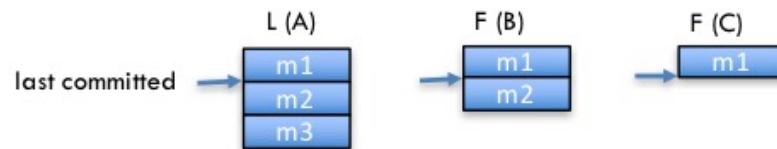


NB: if C had been chosen as leader m2 could also be lost.

3. B commits new messages m4, m5. C follows B.



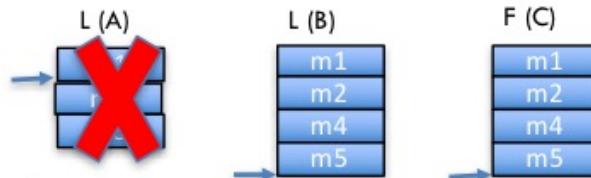
# Example of Replica Recovery (5)



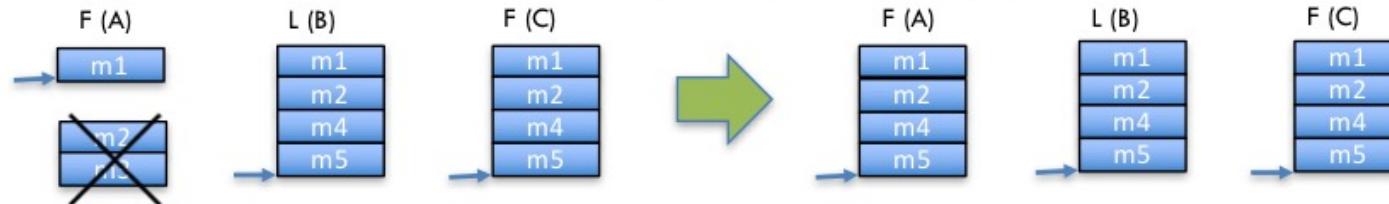
2. A fails and B is new leader; ISR = {B,C}; B commits m2, but not m3. C syncs with new leader B.



3. B commits new messages m4, m5. C follows B.



4. A comes back, truncates to m1 and catches up; finally ISR = {A,B,C}



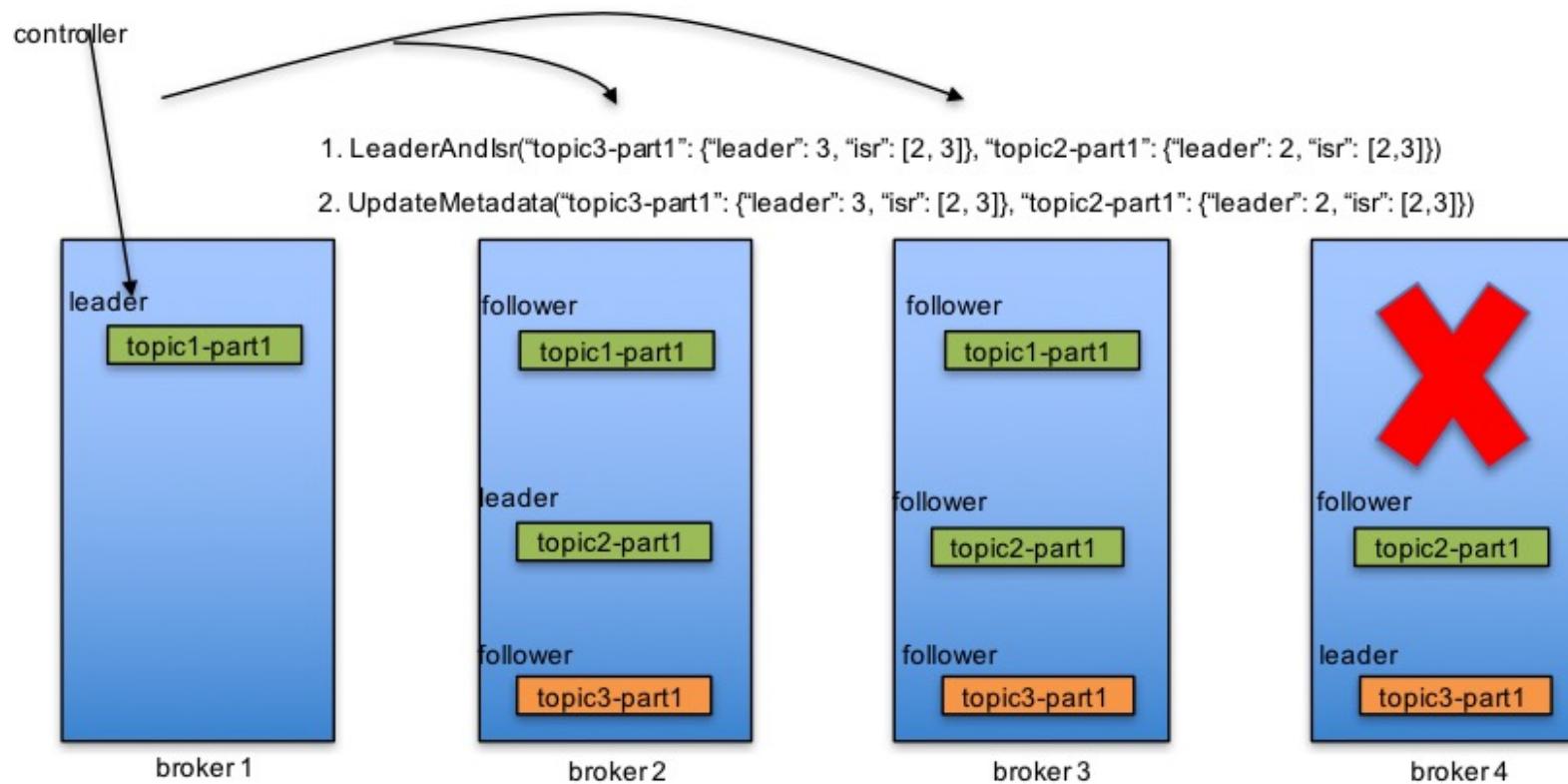
(So m3 is lost – but producer will not have been ack'd if acks=all so will retry)

# Failure Detection and Partition State Management

---

- **One Broker is designated as the Controller to manage the cluster**
  - Detects Broker failure/restart via ZooKeeper
- **Controller action on Broker failure:**
  - Selects a new Leader and updates the ISR
  - Persists the new Leader and ISR to ZooKeeper
  - Sends the new Leader/ISR change to all Brokers
- **Controller action on Broker restart:**
  - Sends Leader and ISR information to the restarted Broker
- **Automated Controller failover**
  - Partition state is recovered from ZooKeeper

# Controller Flow



# Detecting Leader Failure

---

- **zookeeper.session.timeout.ms determines the time before a Leader is considered failed**
  - Set it to be higher than typical Garbage Collection time
  - Default is 6s
  - This is the largest component of rebalancing time

# Detecting Follower Lag

---

- Leader polices ISR by monitoring the Follower lag
  - Setting is `replica.lag.time.max.ms`
  - This setting is important:
    - Controls the lag between Leader and Replica
    - Too large, and slow Replicas will slow down writes
    - Too small, and Replicas will drop in and out of ISR
- Note: in Kafka versions before 0.9, the setting `replica.lag.max.messages` was also available
  - Having two settings was confusing, so it was removed in version 0.9

# Replica Configurations

---

- **default.replication.factor (This is for auto-generated topics)**
  - Default is 1
  - Increase this value for better durability guarantees

# Availability vs Durability

---

- **unclean.leader.election.enable**
  - Determines whether the Kafka Controller should resort to electing a Leader that is not in sync if it has no other choice
    - This could result in data loss
  
- **min.insync.replicas**
  - The minimum number of Replicas that must be in sync (in ISR) to write data
  - Otherwise the Producer will receive a NotEnoughReplicas exception
  - This can be used with acks=all on the Producer to provide stronger write guarantees

# Balancing the Leaders

---

- **Ideally, Leaders should be evenly distributed across all Brokers**
  - Leaders do more work
  - Leaders can change on failure
- **Approach:**
  - First replica is the preferred replica
  - During assignment, spread the preferred replicas evenly among Brokers
  - Periodically move the Leader to the preferred replica, if possible

# Preferred Replica

```
ben$ kafka-topics --zookeeper localhost:2181 --describe --topic i-love-logs
Topic:i-love-logs    PartitionCount:3    ReplicationFactor:3    Configs:
    Topic: i-love-logs    Partition: 0    Leader: 1    Replicas: 1,2,0 Isr: 0,2,1
    Topic: i-love-logs    Partition: 1    Leader: 2    Replicas: 2,0,1 Isr: 2,0,1
    Topic: i-love-logs    Partition: 2    Leader: 0    Replicas: 0,1,2 Isr: 2,0,1
```



- The Preferred Replica helps ensure good load balancing
- Kafka will automatically rebalance to ensure the Preferred Replica is used

# The Kafka Data Files (1)

---

- **Each Broker has a data directory specified in the `server.properties` file**
  - e.g., `log.dirs=/var/lib/kafka_0`
- **Each topic-partition has a separate subdirectory**
  - e.g., `/var/lib/kafka_0/mytopic-0`
- **These contain two types of files:**
  - Index file (maps the virtual to physical offsets)
    - Filename suffix: `.index`
  - Data file (holds the messages and metadata)
    - Filename suffix: `.log`

# The Kafka Data Files (2)

---

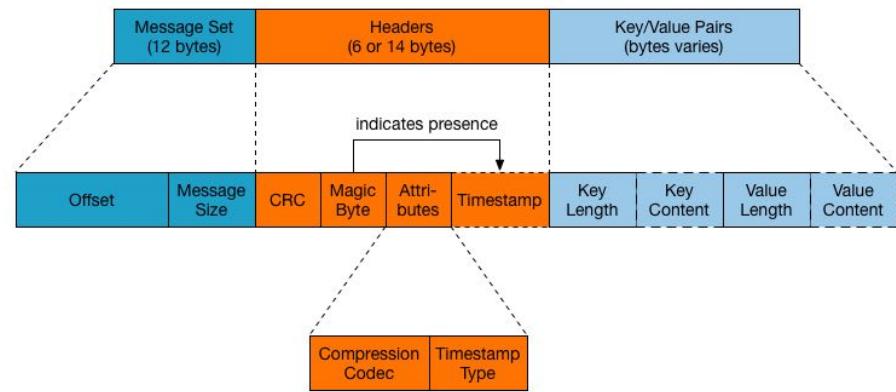
- Each separate log file is a *segment* of the overall log
- A segment is closed, and a new one created, when the first of the following property values is exceeded:
  - `log.roll.ms` or, if that value is not present, `log.roll.hours` (default 168 hours)
  - `log.segment.bytes` (default 1073741824 bytes)
- The filename is the offset of the first message in the segment

# The Kafka Data Files (3)

---

- In addition, each Broker has two checkpoint files:
  - recovery-point-offset-checkpoint
    - The point up to which data has been flushed to disk
    - During recovery, the Broker will only check the validity of messages beyond this point
  - replication-offset-checkpoint
    - The last committed offset
    - On startup, the follower uses this to truncate any uncommitted data

# The Format of a Kafka Message



- <http://kafka.apache.org/documentation.html#messageformat>

# Important JMX Metrics (1)

---

- **ActiveControllerCount**
  - kafka.controller:type=KafkaController,name=ActiveControllerCount
  - Number of active Controllers in the Cluster. Alert if the value is anything other than 1
- **OfflinePartitionsCount**
  - kafka.controller:type=KafkaController,Name=OfflinePartitionsCount
  - Number of Partitions that do not have an active leader and hence are not writable or readable. Alert if the value is greater than 0

# Important JMX Metrics (2)

---

- **UnderReplicatedPartitions**

- kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions
  - Number of under-replicated Partitions ( $| \text{ISR} | < | \text{all replicas} |$ ). Alert if the value is greater than 0

- **PartitionCount**

- kafka.server:type=ReplicaManager, name=PartitionCount
  - Number of Partitions on this Broker. This should be relatively even across all Brokers

# Important JMX Metrics (3)

---

- **LeaderCount**

- kafka.server:type=ReplicaManager, name=LeaderCount
- Number of leaders on this Broker. This should be approximately even across all Brokers. If not, set auto.leader.rebalance.enable to true on all Brokers in the cluster

- **IsrExpandsPerSec**

- kafka.server:type=ReplicaManager, name=IsrExpandsPerSec
- When a Broker is brought up after a failure, it starts catching up by reading from the Leader. Once it is caught up, it gets added back to the ISR

- **IsrShrinksPerSec**

- kafka.server:type=ReplicaManager, name=IsrShrinksPerSec
- If a Broker goes down, ISR for some of the partitions will shrink. When that Broker is up again, ISR will be expanded once the replicas are fully caught up. Other than that, the expected value for both ISR shrink rate and expansion rate is 0

# Important log4j Files

---

- In addition to the Broker's log, pay attention to the following log files:
  - controller.log
    - Logs all Broker failures, and actions taken because of them
  - state-change.log
    - Logs every decision it has received from the Controller

# Quiz

---

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

# Quiz: Answer

---

- ISR fluctuations might explain the pauses, depending on the value of `replica.lag.time`, but the presence of duplicate messages implies leaders are being re-elected
- To investigate:
  - Are we seeing GCs greater than the ZooKeeper timeout? Check ZooKeeper too
  - Check `LeaderElectionsPerSec` metric. This indicates that the Controller is moving leadership due to Leaders becoming unresponsive. The Broker log should show entries like `session expired` and the Broker registering in ZooKeeper
  - Check `ReplicaMaxLag` metric. If this goes beyond `replica.lag.time.max.ms` then Followers will be ejected from the ISR
  - Check `ResponseRemoteTime` metric. This tells you how long messages take to replicate. If there is a spike here, there is something going wrong with replication
  - Correlate with settings for `replica.lag.time.max.ms`

# Intra-Cluster Replication

---

- *Basic Replication Concepts*
- *Replica Recovery and Failure Detection*
- **Chapter Review**

# Chapter Review

---

- Replication provides reliability and high availability for the cluster
- The ISR list contains all In Sync Replicas
- Several JMX metrics are available to help you monitor and troubleshoot replication

# Log Retention and Compaction

Chapter 5



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

**>>> 05: Log Retention and Compaction**

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Log Retention and Compaction

---

- In this chapter you will learn:
  - What log compaction is, and why it is useful
  - Limitations on compact topics

# Log Retention and Compaction

---

- **Log Retention and Compaction**
- *Chapter Review*

## Log Retention

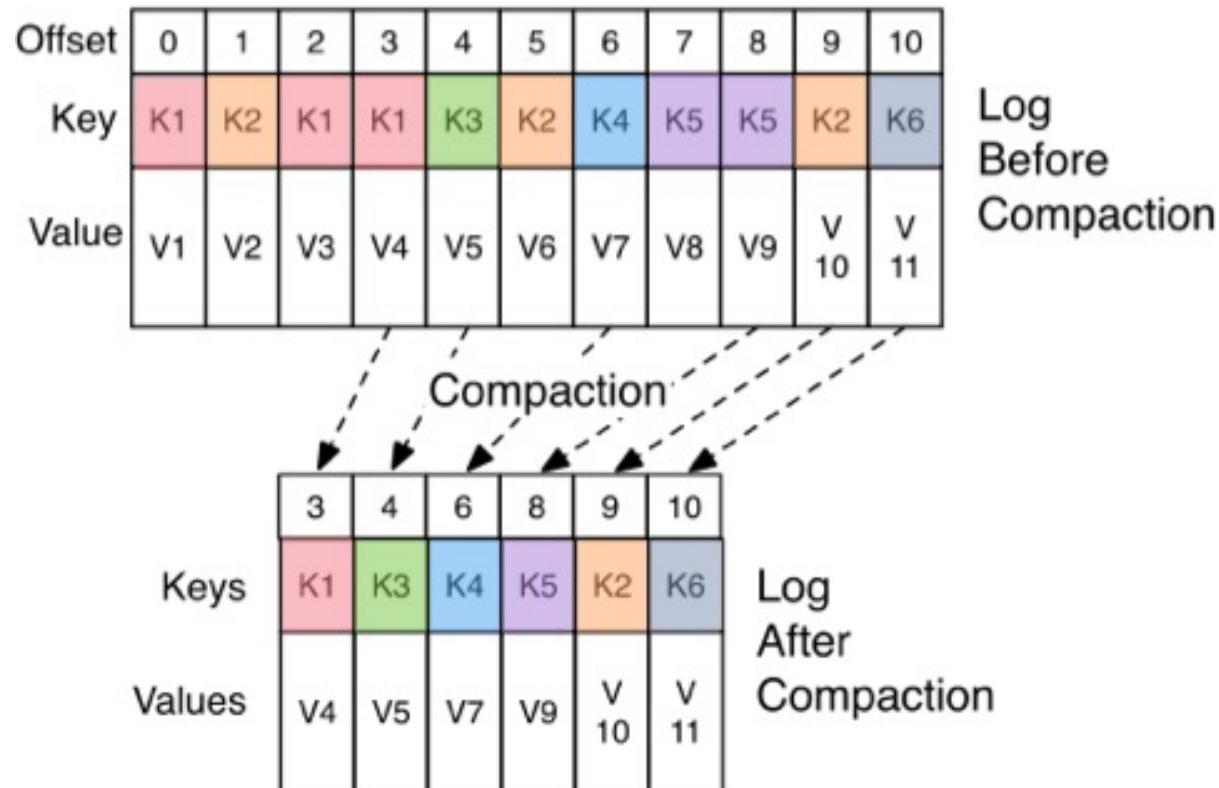
- Duration default: messages will be retained for seven days
- Duration is configurable per broker by changing one of, in order of priority:
  - log.retention.ms
  - log.retention.minutes
  - log.retention.hours
- A topic can override a broker's configured duration with the property retention.ms
- There is no default limit on the size of the log
  - You can set this with retention.bytes

# Log Retention Policy

---

- **log.cleanup.policy**
  - delete: old segments are deleted (default)
    - Removes all log segments where all messages are older than the retention.ms or until the log size is below retention.bytes (or both)
  - compact: old values for the key are deleted (change this to create compacted topics)

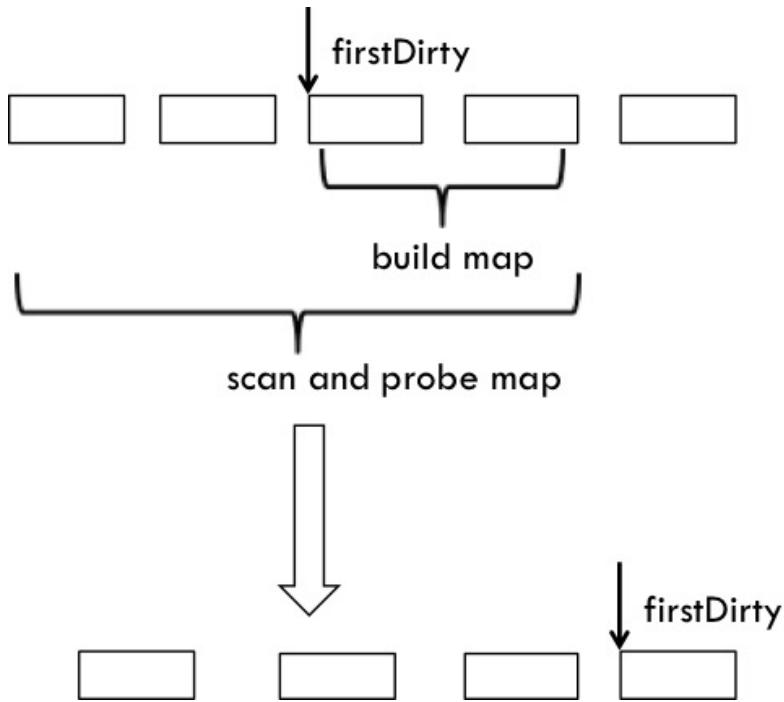
# Log Compaction: What is it?



## ■ Uses:

- Database change capture
- Stateful stream processing
- Event sourcing

# Log Compaction: Implementation



- Build 'dirty map' from dirty log segments
- Create a copy of each clean segment, removing any keys existing in the dirty map
- Switch segment pointer to new copy
- Store clean offset in cleaner-offset-checkpoint file
- Segments are merged if they get too small

# Important Configuration Values

---

- **log.cleaner.min.cleanable.ratio**
  - Default is 0.5
  - Only trigger log clean if the ratio of dirty/total is larger than this value
- **Most recent segment is never cleaned**
  - Make sure segments are rolled
    - log.segment.bytes, log.roll.ms
- **log.cleaner.io.max.bytes.per.second**
  - Default is infinite
  - Can be used for throttling

# Be Careful With Deletes

---

- Delete tombstone modelled as a null message
- Danger of removing a deleted key too soon
  - Consumer still assumes the old value with the key
- `log.cleaner.delete.retention.ms`
  - Default 1 day
  - “Delete tombstone” removed after that time
  - Consumer needs to finish consuming the tombstone before that time

# Useful JMX Metrics

---

- **Max dirty percent**
  - kafka.log:type=LogCleanerManager, name=max-dirty-percent
- **Cleaner recopy percent**
  - kafka.log:type=LogCleaner, name=cleaner-recopy-percent
- **Max clean time**
  - Last cleaning time
  - kafka.log:type=LogCleaner, name=max-clean-time-secs

# Log Messages

---

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

```
Beginning cleaning of log topic1,0
```

```
Building offset map for topic1,0 ...
```

```
Log cleaner thread 0 cleaned log topic1,0 (dirty section=[100111,200011])
```

# Log Retention and Compaction

---

- *Log Retention and Compaction*
- **Chapter Review**

# Chapter Review

---

- Log compaction is useful if only the last value for each key is required

# Developing With Kafka

Chapter 6



# Course Contents

---

- 01: Introduction
- 02: Kafka Fundamentals
- 03: Kafka's Architecture
- 04: Intra-Cluster Replication
- 05: Log Retention and Compaction
- >>> **06: Developing With Kafka**
- 07: More Advanced Kafka Development
- 08: Schema Management In Kafka
- 09: Kafka Connect for Data Movement
- 10: Basic Kafka Administration
- 11: Runtime Configurations
- 12: Monitoring and Alerting
- 13: Kafka Security
- 14: Kafka Streams

# Developing With Kafka

---

- **In this chapter you will learn:**
  - How Kafka developers typically use Maven for application packaging and deployment
  - How to write a Producer using the Java API
  - How to use the REST proxy to access Kafka from other languages
  - How to write a basic Consumer using the New Consumer API

# Developing With Kafka

---

- **Using Maven for Project Management**
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Summary*

# About Maven

---



- **Apache Maven is an open-source software project management tool**
  - Makes it easy to handle programming tasks such as:
    - Downloading and managing dependencies
    - Creating artifacts such as JARs
    - Compilation, and support for continuous integration
    - Creation of project files for IDEs such as Eclipse
    - etc.
- **All Maven configuration is done using XML files known as POM files**
  - (Project Object Model)
- **Functionality can be expanded using plugins**

# Why Use Maven?

---

- **Kafka has many dependencies**
  - This can make things difficult for developers, especially when creating JARs
- **Kafka supports multiple versions of Scala**
  - Maven makes it easy to use the right version
- **Adding new dependencies is very easy**
  - Most Kafka, and Kafka ecosystem, projects support Maven directly

# Sample Maven POM for Kafka (1)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!-- Put your program's details below -->
  <groupId>io.confluent.kafka.program</groupId>
  <artifactId>kafkaprogram</artifactId>

  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <repositories>
    <repository>
      <id>confluent</id>
      <url>http://packages.confluent.io/maven/</url>
    </repository>
  </repositories>
```

# Sample Maven POM for Kafka (2)

```
<properties>
    <!-- Keep Kafka versions as properties to allow easy modification -->
    <kafka.version>0.10.0.0-cp1</kafka.version>
    <!-- Maven properties for compilation -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
<dependencies>
    <!-- Add the Kafka client dependency -->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>${kafka.version}</version>
    </dependency>
</dependencies>
```

# Sample Maven POM for Kafka (3)

```
<build>
  <plugins>
    <plugin>
      <!-- Set the Java target version to 1.8 -->
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.0</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-eclipse-plugin</artifactId>
      <version>2.9</version>
      <configuration>
        <downloadJavadocs>true</downloadJavadocs>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

# Using Maven (1)

---

- To compile a Maven project:

```
mvn compile
```

- To run unit tests:

```
mvn test
```

- To create a JAR artifact

```
mvn package
```

- JAR file will be created in a separate subdirectory
  - By default the directory is named target

# Using Maven (2)

---

- To build and run a project managed by Maven:

```
mvn exec:java -Dexec.mainClass="path.to.MainClass"
```

- If your application requires arguments, use:

```
mvn exec:java -Dexec.mainClass="path.to.MainClass" -Dexec.args="myarguments"
```

# Creating Eclipse Projects with Maven

---

- Maven can be used to create IDE project files
  - Eclipse files are created
  - IntelliJ supports Maven directly
- Add the Maven repositories to Eclipse's path

```
mvn -Dworkspace=/home/training/workspace eclipse:configure-workspace
```

- Create the Eclipse project files for the Maven project

```
mvn eclipse:eclipse
```

- You can then import the project into Eclipse

# Maven And Our Exercise Environment

---

- One common complaint with Maven is that it “downloads the Internet”
  - When first run, it will download all dependencies, and *their* dependencies, and so on
- This can require a significant amount of time, and network bandwidth
- To avoid this during class, we have set Maven to run in offline mode; all the dependencies have already been downloaded

# Developing With Kafka

---

- *Using Maven for Project Management*
- **Programmatically Accessing Kafka**
- *Writing a Producer in Java*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Summary*

# The Kafka API

---

- Recent versions of Kafka include Java clients in the `org.apache.kafka.clients` package
  - These are intended to supplant the older Scala clients
  - They are available in a JAR which has as few dependencies as possible, to reduce code size
- There are client libraries for many other languages
  - The quality and support for these varies
- Confluent supports `librdkafka`, a C/C++ client library
  - Libraries for other languages will be supported in the future
- Confluent also maintains a REST Proxy for Kafka
  - This allows any language to access Kafka via REST
    - (REpresentational State Transfer; essentially, a way to access a system by making HTTP calls)

# Developing With Kafka

---

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- **Writing a Producer in Java**
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Summary*

# The Producer API

---

- To create a Producer, use the KafkaProducer class
- This is thread safe; sharing a single Producer instance across threads will typically be faster than having multiple instances
- Create a Properties object, and pass that to the Producer
  - You will need to specify one or more Broker host/port pairs to establish the initial connection to the Kafka cluster
    - The property for this is bootstrap.servers
    - This is only used to establish the initial connection
    - The client will use all servers, even if they are not all listed here
    - Question: why not just specify a single server?

# Important Properties Elements (1)

Name	Description
<code>bootstrap.servers</code>	List of Broker host/port pairs used to establish the initial connection to the cluster
<code>key.serializer</code>	Class used to serialize the key. Must implement the <b>Serializer</b> interface
<code>value.serializer</code>	Class used to serialize the value. Must implement the <b>Serializer</b> interface
<code>compression.type</code>	How data should be compressed. Values are <code>none</code> , <code>snappy</code> , <code>gzip</code> , <code>lz4</code> . Compression is performed on batches of records

# Important Properties Elements (2)

acks

Number of acknowledgment the Producer requires the Leader to have before considering the request complete. This controls the durability of records. **acks=0**: Producer will not wait for any acknowledgment from the server; **acks=1**: Producer will wait until the Leader has written the record to its local log; **acks=all**: Producer will wait until all in-sync replicas have acknowledged receipt of the record

# Creating the Properties and KafkaProducer Objects

```
1 Properties props = new Properties;  
2 props.put("bootstrap.servers", "broker1:9092");  
3 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
4 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
5  
6 KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

- Other serializers available: `ByteArraySerializer`, `IntegerSerializer`, `LongSerializer`
- `StringSerializer` encoding defaults to `UTF8`
  - Can be customized by setting the property `serializer.encoding`

# Aside: Helper Classes

---

- Kafka includes helper classes `ProducerConfig`, `ConsumerConfig`, and `StreamsConfig` which provide predefined constants for commonly configured properties
- Example:

```
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
```

# Sending Messages to Kafka

```
1 String k = "mykey";
2 String v = "myvalue";
3 ProducerRecord<String, String> record = new ProducerRecord<String, String>("mytopic", k, v); ①
4 producer.send(record);
5
6 producer.close();
```

## ① Alternatively:

```
producer.send(new ProducerRecord<String, String>("mytopic", k, v));
```

# send() Does Not Block

---

- **The send() call is asynchronous**
  - It does not block; it returns immediately after it has added the message to a buffer of pending record sends
    - This allows the Producer to send records in batches for better performance
  - It returns immediately, and your code continues
- **It returns a Future which contains a RecordMetadata object**
  - The Partition the record was put into and its offset
- **To force send() to block, call producer.send(record).get()**

# Retrying Sends

---

- The **retries** property specifies how many times the Producer will attempt to retry sending records for which it has received a transient failure
  - Defaults to 0
  - Only relevant if acks is not 0
- Note that changing this value can result in records being written out of order

# send() and Callbacks (1)

---

- **send(record) is equivalent to send(record, null)**
- **Instead, it is possible to supply a Callback as the second parameter**
  - This is invoked when the send has been acknowledged
  - It is an Interface with an onCompletion method:

```
onCompletion(RecordMetadata rm, java.lang.Exception exception)
```

- **rm will be null if an error occurred**
- **exception will be null if no error occurred**

# send() and Callbacks (2)

- Example code:

```
1 producer.send(record, new Callback() {  
2     public void onCompletion(RecordMetadata metadata, Exception e) {  
3         if(e != null) {  
4             e.printStackTrace();  
5         } else {  
6             System.out.println("The offset of the record we just sent is: " + metadata.offset());  
7         }  
8     }  
9 });
```

# Developing With Kafka

---

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- **Hands-On Exercise: Writing a Producer**
- *Writing a Consumer in Java*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Summary*

# Hands-On Exercise: Writing a Producer

---

- In this Hands-On Exercise, you will write a Kafka Producer
- Please refer to the Hands-On Exercise Manual

# Developing With Kafka

---

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Hands-On Exercise: Writing a Producer*
- **Writing a Consumer in Java**
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Summary*

# The New Consumer API

---

- We will be using the ‘New Consumer’ API
  - Introduced in Kafka 0.9
- Prior to Kafka 0.9, there were two Consumers:
  - ‘High Level Consumer’ and SimpleConsumer
    - The name of the second is misleading!
    - High Level Consumer used ZooKeeper to track offsets
  - The New Consumer in 0.9 combines the features of both older Consumers
    - Also supports security
- New Consumer uses the KafkaConsumer class

# Consumers and Offsets

---

- **Each message in a Partition has an offset**
  - The numerical value indicating where the message is in the log
- **Kafka tracks the Consumer Offset for each partition of a topic the Consumer (or Consumer Group) has subscribed to**
  - It tracks these values in a special topic
- **Consumer offsets are committed automatically by default**
  - We will see later how to manually commit offsets if you need to do that
- **Tip: the Consumer Offset is the value of the next message the Consumer will read, not the last message that has been read**
  - For example, if the Consumer Offset is 9, this indicates that messages 0 to 8 have already been processed, and that message 9 will be the next one sent to the Consumer

# Important Consumer Properties

- Important Consumer properties include:

Name	Description
<code>bootstrap.servers</code>	List of Broker host/port pairs used to establish the initial connection to the cluster
<code>key.deserializer</code>	Class used to deserialize the key. Must implement the <b>Deserializer</b> interface
<code>value.deserializer</code>	Class used to deserialize the value. Must implement the <b>Deserializer</b> interface
<code>group.id</code>	A unique string that identifies the Consumer Group this Consumer belongs to.
<code>enable.auto.commit</code>	When set to <b>true</b> (the default), the Consumer will trigger offset commits based on the value of <code>auto.commit.interval.ms</code> (default 5000ms)

# Creating the Properties and KafkaConsumer Objects

```
1 Properties props = new Properties();
2 props.put("bootstrap.servers", "broker1:9092");
3 props.put("group.id", "samplegroup");
4 props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
5 props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
6
7 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
8 consumer.subscribe(Arrays.asList("my_topic_1", "my_topic_2)); ①
```

- ① The Consumer can subscribe to as many topics as it wishes, although typically this is often just a single topic. Note that this call is not additive; calling subscribe again will remove the existing list of topics, and will only subscribe to those specified in the new call

# Reading Messages from Kafka with poll()

```
1 while (true) { ①
2     ConsumerRecords<String, String> records = consumer.poll(100); ②
3     for (ConsumerRecord<String, String> record : records)
4         System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(),
5             record.value());
6 }
```

① Loop forever

② Each call to poll returns a (possibly empty) list of messages. The parameter controls the maximum amount of time in ms that the Consumer will block if no new records are available. If records are available, it will return immediately.

# Aside: Controlling the Number of Messages Returned

---

- By default, `poll()` returns all available messages
  - Up to the maximum data size *per partition*
    - `max.partition.fetch.bytes`: default value 1048576
  - A topic with many partitions could result in extremely large amounts of data being returned
- In Kafka 0.10.0, `max.poll.records` was introduced
  - Property limits the total number of records retrieved in a single call to `poll()`

# Preventing Resource Leaks

---

- It is good practice to wrap the code in a `try{ }` block, and close the KafkaConsumer object in a `finally{ }` block to avoid resource leaks

```
1 try {  
2     while (true) { ①  
3         ConsumerRecords<String, String> records = consumer.poll(100); ②  
4         for (ConsumerRecord<String, String> record : records)  
5             System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(),  
record.value());  
6     } finally {  
7         consumer.close();  
8     }  
}
```

# Important: The KafkaConsumer Is Not Thread-Safe

---

- It is important to note that KafkaConsumer is not thread-safe
- We will demonstrate how to write a multi-threaded Consumer in the next chapter

# Developing With Kafka

---

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- **Hands-On Exercise: Writing a Basic Consumer**
- *Chapter Summary*

# Hands-On Exercise: Writing a Basic Consumer

---

- In this Hands-On Exercise, you will write a basic Kafka Consumer
- Please refer to the Hands-On Exercise Manual

# Developing With Kafka

---

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Hands-On Exercise: Writing a Basic Consumer*
- **Chapter Summary**

# Chapter Summary

---

- The Kafka API provides Java clients for Producers and Consumers
- Client libraries for other languages are available, though the quality varies
  - Confluent supports a C/C++ client, a Python client, and others are being developed

# More Advanced Kafka Development

Chapter 7



# Course Contents

---

- 01: Introduction
- 02: Kafka Fundamentals
- 03: Kafka's Architecture
- 04: Intra-Cluster Replication
- 05: Log Retention and Compaction
- 06: Developing With Kafka
- >>> 07: More Advanced Kafka Development**
- 08: Schema Management In Kafka
- 09: Kafka Connect for Data Movement
- 10: Basic Kafka Administration
- 11: Runtime Configurations
- 12: Monitoring and Alerting
- 13: Kafka Security
- 14: Kafka Streams

# More Advanced Kafka Development

---

- **In this chapter you will learn:**

- How to write a multi-threaded Consumer in Java
- How to specify the offset to read from
- How to manually commit reads from the Consumer
- How to create a custom Partitioner
- How to control message delivery reliability

# More Advanced Kafka Development

---

- **Creating a Multi-Threaded Consumer**
- *Specifying Offsets*
- *Consumer Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Summary*

# Limitations of our Previous Consumer

---

- The Consumer we wrote in the previous chapter ran in a single thread
- You may want a program to process messages from multiple topics in different ways
  - Or you may have a powerful machine, and want a program to be able to process many messages simultaneously
- KafkaConsumer is not thread-safe, so it is your responsibility to create multiple KafkaConsumer objects, one per thread
- The KafkaConsumer's wakeup() method can be called from an external thread to interrupt the poll
  - It will throw a WakeupException
  - You should catch that exception and call the close() method to free up resources

# Creating a Multi-Threaded Consumer (1)

- We use the Runnable Interface

```
1 public class MyConsumerLoop implements Runnable {  
2     private KafkaConsumer<String, String> consumer;  
3     private List<String> topics;  
4     private int id;  
5  
6     public MyConsumerLoop(int id, String groupId, List<String> topics) {  
7         this.id = id;  
8         this.topics = topics;  
9         Properties props = new Properties();  
10        props.put("bootstrap.servers", "broker1:9092");  
11        props.put("group.id", groupId);  
12        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
13        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
14        this.consumer = new KafkaConsumer<>(props);  
15    }  
}
```

# Creating a Multi-Threaded Consumer (2)

```
1 public void run() {  
2     try {  
3         consumer.subscribe(topics);  
4  
5         while (true) {  
6             ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE); ①  
7             for (ConsumerRecord<String, String> record : records) {  
8                 // Do something with the message!  
9             }  
10        }  
11    } catch (WakeupException e) {  
12        // ignore for shutdown  
13    } finally {  
14        consumer.close();  
15    }  
16 }  
17  
18 public void shutdown() {  
19     consumer.wakeup();  
20 }  
21 }
```

- ① This will cause poll to block indefinitely

# Creating a Multi-Threaded Consumer (3)

- Now instantiate five threads and start them

```
1 public static void main(String[] args) {  
2     int numConsumers = 5;  
3     String groupId = "my-consumer-group";  
4     List<String> topics = Arrays.asList("mytopic");  
5     ExecutorService executor = Executors.newFixedThreadPool(numConsumers);  
6  
7     List<MyConsumerLoop> consumers = new ArrayList<>();  
8     for (int i = 0; i < numConsumers; i++) {  
9         MyConsumerLoop consumer = new MyConsumerLoop(i, groupId, topics);  
10        consumers.add(consumer);  
11        executor.submit(consumer);  
12    }  
}
```

# Creating a Multi-Threaded Consumer (4)

- Finally, configure what to do when the process terminates

```
1 Runtime.getRuntime().addShutdownHook(new Thread() {
2     @Override
3     public void run() {
4         for (MyConsumerLoop consumer : consumers) {
5             consumer.shutdown();
6         }
7         executor.shutdown();
8         try {
9             executor.awaitTermination(5000, TimeUnit.MILLISECONDS);
10        } catch (InterruptedException e) {
11            e.printStackTrace();
12        }
13    }
14 });
15 }
```

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- **Specifying Offsets**
- *Consumer Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Summary*

# Determining the Offset When a Consumer Starts

---

- The `Consumer` property `auto.offset.reset` determines what to do if there is no legal offset in Kafka for the Consumer's Consumer Group
  - The first time a particular Consumer Group starts
  - If the offset points to a nonexistent record
- The value can be one of:
  - earliest: Automatically reset the offset to the earliest available
  - latest: Automatically reset to the latest offset available
  - none: Throw an exception if no previous offset can be found for the ConsumerGroup
- The default is latest

# Changing the Offset Within the Consumer (1)

---

- The KafkaConsumer API provides a way to dynamically change the offset from which the Consumer will read
  - And to view the current offset
- `position(TopicPartition)` provides the offset of the next record that will be fetched
- `seekToBeginning(Collection<TopicPartition>)` seeks to the first offset of each of the specified Partitions
- `seekToEnd(Collection<TopicPartition>)` seeks to the last offset of each of the specified Partitions
- `seek(TopicPartition, offset)` seeks to a specific offset in the specified Partition

# Changing the Offset Within the Consumer (2)

- For example, to seek to the beginning of all partitions that are being read by a Consumer for a particular topic, you might do something like:

```
1 ...
2 consumer.subscribe("mytopic");
3 consumer.poll(0);
4 consumer.seekToBeginning(consumer.assignment()); ①
```

- ① **assignment()** returns a list of all TopicPartitions currently assigned to this Consumer

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- *Specifying Offsets*
- **Consumer Rebalancing**
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Summary*

# Adding Consumers Will Cause Rebalancing

---

- So far, we have said that all the data from a particular Partition will go to the same Consumer
- This is true, as long as more Consumers are not added to the Consumer Group
  - And as long as Consumers in the Consumer Group do not fail
- If the number of Consumers changes, a partition rebalance occurs
  - Partition ownership is moved around between the Consumers to spread the load evenly
  - Consumers cannot consume messages during the rebalance, so this results in a short pause in message consumption

# The Case For and Against Rebalancing

---

- **Typically, partition rebalancing is a good thing**
  - It allows you to add more Consumers to a Consumer Group dynamically, without having to restart all the other Consumers in the group
  - It automatically handles situations where one Consumer in the Consumer Group fails
- **However, if your Consumer is relying on getting all data from a particular Partition, this could be a problem**
  - One solution: Only have a single Consumer for the entire topic
    - Downside: Lacks scalability
  - An alternative is to provide a ConsumerRebalanceListener when calling subscribe()
    - You implement onPartitionsRevoked and onPartitionsAssigned methods

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- *Specifying Offsets*
- *Consumer Rebalancing*
- **Manually Committing Offsets**
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Summary*

# Default Behavior: Automatically Committed Offsets

---

- By default, `enable.auto.commit` is set to true
  - Consumer offsets are periodically committed in the background
    - This happens during the `poll()` call
- This is typically the desired behavior
- However, there are times when it may be problematic

# Problems With Automatically Committed Offsets

---

- By default, automatic commits occur every five seconds
- Imagine that two seconds after the most recent commit, a rebalance is triggered
  - After the rebalance, Consumers will start consuming from the latest committed offset position
- In this case, the offset is two seconds old, so all messages that arrived in those two seconds will be processed twice
  - This provides “at least once” delivery

# Manually Committing Offsets

---

- A Consumer can manually commit offsets to control the committed position
  - Disable automatic commits: set `enable.auto.commit` to `false`
- `commitSync()`
  - Blocks until it succeeds, retrying as long as it does not receive a fatal error
  - For “at most once” delivery, call `commitSync()` immediately after `poll()` and then process the messages
  - Consumer should ensure it has processed all the records returned by `poll()` or it may miss messages
- `commitAsync()`
  - Returns immediately
  - Optionally takes a callback that will be triggered when the Broker responds
  - Has higher throughput since the Consumer can process next message batch before commit returns
    - Trade-off is the Consumer may find out later the commit failed

# Aside: What Offset is Committed?

---

- Note: The offset committed (whether automatically or manually) is the offset of the next record to be read
  - Not the offset of the last record which was read

# Storing Offsets Outside of Kafka

---

- By default, Kafka stores offsets in a special topic
  - Called \_\_consumer\_offsets
- In some cases, you may want to store offsets outside of Kafka
  - For example, in a database table
- If you do this, you can read the value and then use seek() to move to the correct position when your application launches

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- *Specifying Offsets*
- *Consumer Rebalancing*
- *Manually Committing Offsets*
- **Partitioning Data**
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Summary*

# Kafka's Default Partitioning Scheme

---

- Recall that by default, if a message has a key, Kafka will hash the key and use the result to map the message to a specific Partition
  - (Aside: Kafka uses its own hash algorithm, so this will not change if the version of Java on the machine is upgraded and a new hashing algorithm is introduced)
- This means that all messages with the same key will go to the same Partition
- If the key is null and the default Partitioner is used, the record will be sent to a random partition (using a round-robin algorithm)
- You may wish to override this behavior and provide your own Partitioning scheme
  - For example, if you will have many messages with a particular key, you may want one Partition to hold just those messages

# Creating a Custom Partitioner

---

- To create a custom Partitioner, you should implement the Partitioner interface
  - This interface includes configure, close, and partition methods, although often you will only implement partition
  - partition is given the topic, key, serialized key, value, serialized value, and cluster metadata
- It should return the number of the partition this particular message should be sent to (0-based)

# Custom Partitioner: Example

- Assume we want to store all messages with a particular key in one Partition, and distribute all other messages across the remaining Partitions

```
1 public class MyPartitioner implements Partitioner {  
2     public void configure(Map<String, ?> configs) {}  
3     public void close() {}  
4  
5     public int partition(String topic, Object key, byte[] keyBytes,  
6                           Object value, byte[] valueBytes, Cluster cluster) {  
7         List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);  
8         int numPartitions = partitions.size();  
9  
10        if ((keyBytes == null) || !(key instanceof String))  
11            throw new InvalidRecordException("Record did not have a string Key");  
12  
13        if (((String) key).equals("OurBigKey")) ①  
14            return 0; // This key will always go to Partition 0  
15  
16        // Other records will go to the rest of the partitions using a hashing function  
17        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;  
18    }  
19 }
```

- ① This is the key we want to store in its own partition

# An Alternative to a Custom Partitioner

---

- It is also possible to specify the Partition to which a message should be written when creating the ProducerRecord
  - `ProducerRecord<String, String> record = new ProducerRecord<String, String>("the_topic", 0, key, value);`
  - Will write the message to Partition 0
- Discussion: Which method is preferable?

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- *Specifying Offsets*
- *Consumer Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- **Message Durability**
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Summary*

# Replication Factor Affects Message Durability

---

- Recall that topics can be replicated for durability
- Default replication factor is 1
  - Can be specified when a topic is first created, or modified later

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- *Specifying Offsets*
- *Consumer Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- **Hands-On Exercise: Accessing Previous Data**
- *Chapter Summary*

# Hands-On Exercise: Accessing Previous Data

---

- In this Hands-On Exercise you will create a Consumer which will access data already stored in the cluster.
- Please refer to the Hands-On Exercise Manual

# More Advanced Kafka Development

---

- *Creating a Multi-Threaded Consumer*
- *Specifying Offsets*
- *Consumer Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- **Chapter Summary**

# Chapter Summary

---

- Although KafkaConsumer is not thread-safe, it is relatively easy to write a multi-threaded Consumer
- Your Consumer can move through the data in the Cluster, reading from the beginning, the end, or any point in between
- You may need to take Consumer Rebalancing into account when you write your code
- It is possible to specify your own Partitioner if Kafka's default is not sufficient for your needs

# Schema Management In Kafka

Chapter 8



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

**>>> 08: Schema Management In Kafka**

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Schema Management In Kafka

---

- **In this chapter you will learn:**

- What Avro is, and how it can be used for data with a changing schema
- How to write Avro messages to Kafka
- How to use the Schema Registry for better performance

# Schema Management In Kafka

---

- **An Introduction to Avro**
- *Avro Schemas*
- *The Schema Registry*
- *Hands-On Exercise: Using Kafka with Avro*
- *Chapter Summary*

# What is Serialization?

---

- **Serialization is a way of representing data in memory as a series of bytes**
  - Needed to transfer data across the network, or store it on disk
- **Deserialization is the process of converting the stream of bytes back into the data object**
- **Java provides the Serializable package to support serialization**
  - Kafka has its own serialization classes in  
org.apache.kafka.common.serialization
- **Backward compatibility and support for multiple languages are a challenge for any serialization system**

# The Need for a More Complex Serialization System

---

- So far, all our data has been plain text
- This has several advantages, including:
  - Excellent support across virtually every programming language
  - Easy to inspect files for debugging
- However, plain text also has disadvantages:
  - Data is not stored efficiently
  - Non-text data must be converted to strings
    - No type checking is performed
    - It is inefficient to convert binary data to strings

# Avro: An Efficient Data Serialization System

---



- **Avro is an Apache open source project**
  - Created by Doug Cutting, the creator of Hadoop
- **Provides data serialization**
- **Data is defined with a self-describing schema**
- **Supported by many programming languages, including Java**
- **Provides a data structure format**
- **Supports code generation of data types**
- **Provides a container file format**
- **Avro data is binary, so stores data efficiently**
- **Type checking is performed at write time**

# Schema Management In Kafka

---

- *An Introduction to Avro*
- **Avro Schemas**
- *The Schema Registry*
- *Hands-On Exercise: Using Kafka with Avro*
- *Chapter Summary*

# Avro Schemas

---

- Avro schemas define the structure of your data
- Schemas are represented in JSON format
- Avro has three different ways of creating records:
  - Generic
    - Write code to map each schema field to a field in your object
  - Reflection
    - Generate a schema from an existing Java class
  - Specific
    - Generate a Java class from your schema
    - This is the most common way to use Avro classes

# Avro Data Types (Simple)

- Avro supports several simple and complex data types
  - Following are the most common

Name	Description	Java equivalent
<b>boolean</b>	True or false	<b>boolean</b>
<b>int</b>	32-bit signed integer	<b>int</b>
<b>long</b>	64-bit signed integer	<b>long</b>
<b>float</b>	Single-precision floating-point number	<b>float</b>
<b>double</b>	Double-precision floating-point number	<b>double</b>
<b>string</b>	Sequence of Unicode characters	<b>java.lang.CharSequence</b>
<b>bytes</b>	Sequence of bytes	<b>java.nio.ByteBuffer</b>
<b>null</b>	The absence of a value	<b>null</b>

# Avro Data Types (Complex)

Name	Description
enum	A specified set of values
union	Exactly one value from a specified set of types
array	Zero or more values, each of the same type
map	Set of key/value pairs; key is always a <b>string</b> , value is the specified type
fixed	A fixed number of bytes
record	A user-defined field comprising one or more named fields

- record is the most important of these, as we will see

# Example Avro Schema (1)

```
{  
  "namespace": "model",  
  "type": "record",  
  "name": "SimpleCard",  
  "fields": [  
    {  
      "name": "suit",  
      "type": "string",  
      "doc": "The suit of the card"  
    },  
    {  
      "name": "card",  
      "type": "string",  
      "doc": "The card number"  
    }  
  ]  
}
```

# Example Avro Schema (2)

---

- By default, the schema definition is placed in `src/main/avro`
  - File extension is `.avsc`
- The namespace is the Java package name, which you will import into your code
- `doc` allows you to place comments in the schema definition

# Example Schema Snippet with array and map

```
{  
  "name": "cards_list",  
  "type" : {  
    "type" : "array",  
    "items": "string"  
  },  
  "doc" : "The cards played"  
,  
{  
  "name": "cards_map",  
  "type" : {  
    "type" : "map",  
    "values": "string"  
  },  
  "doc" : "The cards played"  
,
```

# Example Schema Snippet with enum

```
{  
  "name": "suit_type",  
  "type" : {  
    "type" : "enum",  
    "name" : "Suit",  
    "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]  
  },  
  "doc" : "The suit of the card"  
},
```

# Schema Evolution

---

- Often, Avro schemas evolve as updates to code happen
- We often want compatibility between schemas
  - Backward compatibility
    - Code with a new version of the schema can read data written in the old schema
    - Code that reads data written with the schema will assume default values if fields are not provided
  - Forward compatibility
    - Code with previous versions of the schema can read data written in a new schema
    - Code that reads data written with the schema ignores new fields
  - Full compatibility
    - Forward and Backward

# Different Schemas in the Same Topic

---

- Different schemas in the same topic can be Backward, Forward, or Full compatible
  - Default is BACKWARD
- If they are neither, set compatibility to NONE
  - Code has no assumptions on schema as long as it is valid Avro
  - Code has full burden to read and process data
- Configuring compatibility
  - Use Rest API

```
$ curl -X PUT -i -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"compatibility": "NONE"}' \
http://localhost:8081/config/topic_name
```

# Integrating Avro Into Your Maven Project (1)

- To use Avro with your Maven project:

```
<!-- Add the Avro dependency -->
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro</artifactId>
  <version>${avro.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-tools</artifactId>
  <version>${avro.version}</version>
</dependency>
```

# Integrating Avro Into Your Maven Project (2)

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>${avro.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>
        <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

# Schema Management In Kafka

---

- *An Introduction to Avro*
- *Avro Schemas*
- **The Schema Registry**
- *Hands-On Exercise: Using Kafka with Avro*
- *Chapter Summary*

# What Is the Schema Registry?

---

- Submitting the Avro schema with each Producer request would be inefficient
- Instead, the Schema Registry allows you to submit a schema and, in the background, returns a schema ID which is used in subsequent Produce and Consume requests
  - It stores schema information in a special Kafka topic
- It also copes with schema evolution; it checks schemas as data is written and read, and throws an exception if the data does not conform to the schema
- The Schema Registry is accessible both via a REST API and a Java API
  - There are also command-line tools, kafka-avro-console-producer and kafka-avro-console-consumer

# Java Avro Producer example

```
1 Properties props = new Properties();
2 props.put("bootstrap.servers", "broker1:9092");
3 // Configure serializer classes
4 props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
5           io.confluent.kafka.serializers.KafkaAvroSerializer.class);
6 props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
7           io.confluent.kafka.serializers.KafkaAvroSerializer.class);
8 // Configure schema repository server
9 props.put("schema.registry.url", "http://schemaregistry1:8081");
10 // Create the producer expecting Avro objects
11 KafkaProducer<Object, Object> avroProducer = new KafkaProducer<Object, Object>(props);
12 // Create the Avro objects for the key and value
13 CardSuit suit = new CardSuit("spades");
14 SimpleCard card = new SimpleCard("spades", "ace");
15 // Create the ProducerRecord with the Avro objects and send them
16 ProducerRecord<Object, Object> record = new
17 ProducerRecord<Object, Object>(
18     "my_avro_topic", suit, card);
19 avroProducer.send(record);
```

# Java Avro Consumer Example

```
1 public class CardConsumer {
2     public static void main(String[] args) {
3         Properties props = new Properties();
4         props.put("bootstrap.servers", "localhost:9092");
5         props.put("group.id", "testgroup");
6         props.put("enable.auto.commit", "true");
7         props.put("auto.commit.interval.ms", "1000");
8         props.put("session.timeout.ms", "30000");
9         props.put("key.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
10        props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
11        props.put("schema.registry.url", "http://schemaregistry1:8081");
12        props.put("specific.avro.reader", "true");
13
14        KafkaConsumer<CardSuit, SimpleCard> consumer = new KafkaConsumer<>(props);
15        consumer.subscribe(Arrays.asList("my_avro_topic"));
16
17        while (true) {
18            ConsumerRecords<CardSuit, SimpleCard> records = consumer.poll(100);
19            for (ConsumerRecord<CardSuit, SimpleCard> record : records) {
20                System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.
key().getSuit(), record.value().getCard());
21
22            }
23        }
24    }
25 }
```

# Command-line Consumer Example

---

```
$ kafka-console-consumer --bootstrap-server broker1:9092 \  
--new-consumer --from-beginning --topic my_avro_topic
```

# Schema Management In Kafka

---

- *An Introduction to Avro*
- *Avro Schemas*
- *The Schema Registry*
- **Hands-On Exercise: Using Kafka with Avro**
- *Chapter Summary*

# Hands-On Exercise: Using Kafka with Avro

---

- In this Hands-On Exercise, you will write and read Kafka data with Avro
- Please refer to the Hands-On Exercise Manual

# Schema Management In Kafka

---

- *An Introduction to Avro*
- *Avro Schemas*
- *The Schema Registry*
- *Hands-On Exercise: Using Kafka with Avro*
- **Chapter Summary**

# Chapter Summary

---

- Using a serialization format such as Avro makes sense for complex data
- The Schema Registry makes it easy to efficiently write and read Avro data to and from Kafka by centrally storing the schema

# Kafka Connect for Data Movement

Chapter 9



# Course Contents

---

- 01: Introduction
- 02: Kafka Fundamentals
- 03: Kafka's Architecture
- 04: Intra-Cluster Replication
- 05: Log Retention and Compaction
- 06: Developing With Kafka
- 07: More Advanced Kafka Development
- 08: Schema Management In Kafka
- >> 09: Kafka Connect for Data Movement
- 10: Basic Kafka Administration
- 11: Runtime Configurations
- 12: Monitoring and Alerting
- 13: Kafka Security
- 14: Kafka Streams

# Kafka Connect for Data Movement

---

- In this chapter you will learn:
  - The motivation for Kafka Connect
  - How to configure Kafka Connect
  - What Sources and Sinks are available

# Kafka Connect for Data Movement

---

- **The Motivation for Kafka Connect**
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# What is Kafka Connect?

---

- Kafka Connect is a system to reliably stream data between Kafka and other data systems
- Kafka Connect is
  - Distributed
  - Scalable
  - Fault-tolerant
- Kafka Connect is open source, and is part of the Apache Kafka distribution

# Why Not Just Use Producers and Consumers?

---

- Why not just write your own application using Kafka Producers and Consumers?
- Advantages of Kafka Connect:
  - Just requires configuration files
    - No coding needed
  - Off-the-shelf, tested plugins are available for a number of common data sources and sinks
  - Has a distributed mode with automatic load balancing
  - Is fault tolerant: provides automatic offset tracking and recovery
  - Is a pluggable/extendable system

# Sample Use-Cases

---

- **Sample use-cases for Kafka Connect:**

- Stream an entire SQL database, or just specific tables, into Kafka
- Stream data from Kafka topics into HDFS (the Hadoop Distributed File System) for batch processing
- Stream data from Kafka topics into Elasticsearch for secondary indexing
- Stream data from Kafka topics into Cassandra
- ...

# Kafka Connect for Data Movement

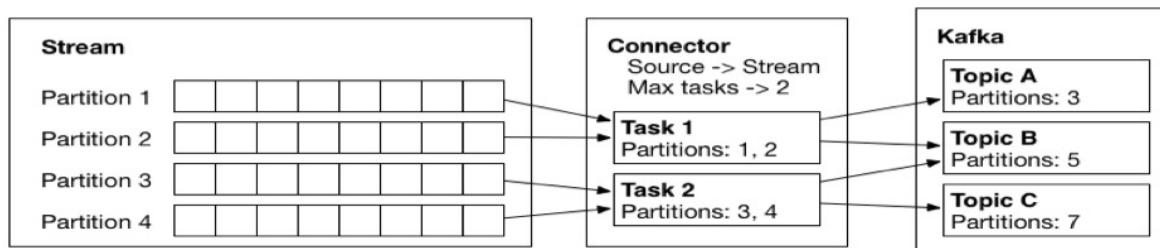
---

- *The Motivation for Kafka Connect*
- **Kafka Connect Basics**
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Kafka Connect Terminology (1)

- Kafka Connect has three main components:

- Connectors
  - Sources and Sinks
- Tasks
- Workers



# Kafka Connect Terminology (2)

---

- **Sources** read data *from* an external data source
- **Sinks** write data *to* an external data source
- **Tasks** are individual pieces of work
  - For example, reading from a particular database table
- **Workers** are processes running tasks
  - One Worker can run multiple tasks in different threads

# Kafka Connect Terminology (3)

---

- Source Connectors



- Sink Connectors



# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- **Modes of Working: Standalone and Distributed**
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Standalone and Distributed Modes

---

- Kafka Connect can be run in two modes
- **Standalone mode**
  - A single process, running on a single machine
- **Distributed mode**
  - Multiple processes, running on multiple machines
    - (Or a single machine, though this is less common)

# Running in Standalone Mode

---

- To run in standalone mode, start a process with one or more connector configurations

```
connect-standalone \
connect-standalone.properties \
connector1.properties \
[connector2.properties connector3.properties...]
```

- Each Connector instance will run in its own thread

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- **Hands-On Exercise: Running in Standalone Mode**
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Hands-On Exercise: Running in Standalone Mode

---

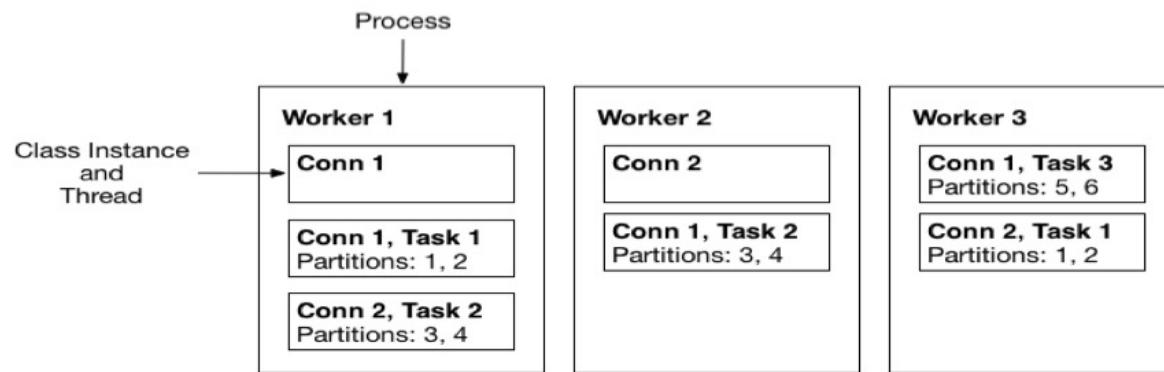
- In this Hands-On Exercise, you will run Kafka Connect in Standalone Mode
- Please refer to the Hands-On Exercise Manual

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- **Distributed Mode**
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Workers in Distributed Mode



# Running in Distributed Mode

---

- To run in distributed mode, start Kafka Connect on each node

```
connect-distributed worker.properties
```

- Connectors can be added, modified, and deleted via a REST API

# Managing Distributed Mode

---

- **To install a new Connector:**
  - Package the Connector in a JAR file
  - Place the JAR in the CLASSPATH of each worker process
- **To update, modify, or delete connectors, use the REST API on port 8083**
  - The REST requests can be made to any worker

# The REST API

- A subset of the REST API is shown below

Method	Path	Description
GET	/connectors	Get a list of active connectors
POST	/connectors	Create a new Connector
GET	/connectors/(string: name)/config	Get configuration information for a Connector
PUT	/connectors/(string: name)/config	Create a new Connector, or update the configuration of an existing Connector

- More information on the REST API can be found at <http://docs.confluent.io>

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- **Hands-On Exercise: Using the Kafka Connect REST API**
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Hands-On Exercise: Using the Kafka Connect REST API

---

- In this Hands-On Exercise, you will query the Kafka Connect REST API
- Please refer to the Hands-On Exercise Manual

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- **Tracking Offsets**
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Tracking Source and Sink Offsets (1)

---

- Kafka Connect tracks the produced and consumed offsets so it can restart tasks at the correct place after a failure
- The method of tracking the offset depends on the specific Connector

# Tracking Source and Sink Offsets (2)

---

- Examples of source and sink offset tracking methods:
  - Standalone local file source
    - A separate local file
  - JDBC source
    - A special Kafka topic (see later)
  - HDFS Sink
    - An HDFS file (see later)

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- **Connector Configuration**
- *Comparing Kafka Connect with Other Options*
- *Chapter Summary*

# Connector Configuration (1)

---

- **Features to consider when configuring Connectors:**

- Distributed mode vs Standalone mode
- Connectors
- Workers
- Overriding Producer and Consumer settings

# Connector Configuration (2)

- **Standalone configuration settings are stored in a file**
  - Filename is specified as a command-line argument
- **Distributed configuration is set via the REST API, in a JSON payload**
- **Connect configuration parameters are as follows:**

Parameter	Description
<code>name</code>	Connector's unique name
<code>connector.class</code>	Connector's Java class
<code>tasks.max</code>	Maximum tasks to create. The Connector may create fewer if it cannot achieve this level of parallelism
<code>topics</code> (Sink connectors only)	List of input topics (to consume from)

# Configuring Workers

---

- Worker configuration is specified in a file which is passed as an argument to the script starting Kafka Connect
- See <http://docs.confluent.io> for a comprehensive list, along with an example configuration file

# Configuring Workers: Both Modes

- Important configuration options common to all workers:

Parameter	Description
<code>bootstrap.servers</code>	A list of host/port pairs to use to establish the initial connection to the Kafka cluster
<code>key.converter</code>	Converter class for the key
<code>value.converter</code>	Converter class for the value

# Configuring Workers: Standalone Mode

Parameter	Description
<b>offset.storage.file.filename</b>	The file in which to store the Connector offsets

# Configuring Workers: Distributed Mode

Parameter	Description
<b>group.id</b>	A unique string that identifies the Kafka Connect cluster group the worker belongs to
<b>config.storage.topic</b>	The topic in which to store Connector and task configuration data. This must be the same for all workers with the same <b>group.id</b>
<b>offset.storage.topic</b>	The topic in which to store offset data for the Connectors. This must be the same for all workers with the same <b>group.id</b>
<b>session.timeout.ms</b>	The timeout used to detect failures when using Kafka's group management facilities
<b>heartbeat.interval.ms</b>	The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Must be smaller than <b>session.timeout.ms</b>

# Off-The-Shelf Connectors

---

- **Connectors included in Confluent Platform:**
  - Local file Source and Sink
  - JDBC Source
  - HDFS Sink
- **Many other connectors are available**
  - See <http://confluent.io/connectors> for a full list

# Local File Source and Sink Connector

---

- Local file Source Connector: tails local file and sends each line as a Kafka message
- Local file Sink Connector: Appends Kafka messages to a local file
- These Connectors work in Standalone mode only

# JDBC Source Connector: Overview

---

- JDBC Source periodically polls a relational database for new or recently modified rows
  - Creates an Avro record for each row, and Produces that record as a Kafka message
- Records from each table are Produced to their own Kafka topic
- New and deleted tables are handled automatically

# JDBC Source Connector: Detecting New and Updated Rows

- The Connector can detect new and updated rows in one of several ways:

Incremental query mode	Description
Incrementing column	Check a single column where newer rows have a larger, autoincremented ID. Does not support updated rows
Timestamp column	Checks a single 'last modified' column. Can't guarantee reading all updates
Timestamp and incrementing column	Combination of the two methods above. Guarantees that all updates are read
Custom query	Used in conjunction with the options above for custom filtering

- Alternative: bulk mode for one-time load, not incremental, unfiltered

# HDFS Sink Connector

---

- Continuously polls from Kafka and writes to HDFS (Hadoop Distributed File System)
- HDFS data format is customizable
  - Avro, Parquet
  - Supports a pluggable partitioner (e.g., based on timestamp)
- Integrates with Hive
  - Auto table creation
  - Schema evolution with Avro
- Provides exactly once delivery
- Supports secure HDFS and Hive Metastore
- See the documentation for a full list of configuration parameters

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- **Comparing Kafka Connect with Other Options**
- *Chapter Summary*

# Kafka Connect vs Alternative Options (1)

---

- There are three categories of systems similar to Connect:
  - Log and metric collection, processing, and aggregation
    - e.g., Flume, Logstash, Fluentd, Heka
  - ETL tools
    - e.g., Morphlines, Informatica
  - Data pipeline management
    - e.g., NiFi

# Kafka Connect vs Alternative Options (2)

	Kafka Connect	Log and Metric Collection	ETL for Data Warehousing	Data Pipeline Management
<b><i>Minimal configuration overhead</i></b>	Yes	Yes	No	Yes
<b><i>Support for stream processing</i></b>	Yes	Lacking	No	Yes
<b><i>Support for batch processing</i></b>	Yes	Lacking	Yes	Yes
<b><i>Focus on reliably and scalably copying data</i></b>	Yes	Broader focus	No	No

# Kafka Connect vs Alternative Options (3)

	Kafka Connect	Log and Metric Collection	ETL for Data Warehousing	Data Pipeline Management
<b>Parallel</b>	Automated	Manual	Automated	Yes
<b>Accessible connector API</b>	Yes	Complex	Narrow	Yes
<b>Scales across multiple systems</b>	Yes	Yes	Yes	Improving

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Kafka Connect Basics*
- *Modes of Working: Standalone and Distributed*
- *Hands-On Exercise: Running in Standalone Mode*
- *Distributed Mode*
- *Hands-On Exercise: Using the Kafka Connect REST API*
- *Tracking Offsets*
- *Connector Configuration*
- *Comparing Kafka Connect with Other Options*
- **Chapter Summary**

# Chapter Summary

---

- Kafka Connect provides a scalable, reliable way to transfer data from external systems into Kafka, and vice versa
- Stock Connectors are provided, and many others are currently under development by Confluent and third parties

# Basic Kafka Administration

Chapter 10



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

**>>> 10: Basic Kafka Administration**

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Basic Kafka Administration

---

- In this chapter you will learn:
  - How to perform some common Kafka administrative tasks

# Basic Kafka Administration

---

- **Administering Kafka**
- *Chapter Summary*

# Configuring Topics

- By default, topics are automatically created when they are first used by a client
  - Replication factor of one, a single partition
  - In production environments, consider disabling automatic topic generation with the `auto.create.topics.enable` boolean
- Alternatively, you can create a topic manually
  - This allows you to set the replication factor and number of partitions

```
$ kafka-topics --zookeeper zk_host:port \
--create --topic topic_name \
--partitions 6 --replication-factor 3
```

- You can also modify topics

```
$ kafka-topics --zookeeper zk_host:port \
--alter --topic topic_name --partitions 40
```

- No data is moved from existing topics
- Changing the number of partitions could cause problems for your application logic!

# Deleting a Topic

---

- It is possible to delete a topic:

```
$ kafka-topics --zookeeper zk_host:port \
--delete --topic topic_name
```

- Note that all Brokers must have the `delete.topic.enable` parameter set to `true`
  - It is false by default
  - If this is not set, the delete command will be silently ignored
- All Brokers must be running for the delete to be successful

# Compressing Data

---

- **Compression can be configured on the Producer**
- **Compression is end-to-end**
  - Compressed on the Producer, stored in compressed format on the Broker, decompressed on the Consumer
- **Specify by setting compression.type**
  - gzip, snappy, or lz4
  - GZip is computationally expensive compared to Snappy and LZ4

# Basic Kafka Administration

---

- *Administering Kafka*
- **Chapter Summary**

# Chapter Summary

---

- Topics can be manually created, modified, and deleted

# Runtime Configurations

Chapter 11



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

>>> 11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Runtime Configurations

---

- In this chapter you will learn:
  - What common configuration settings you might need to change to optimize performance

# Runtime Configurations

---

- Runtime Configuration Settings
- *Chapter Review*

# JVM Settings

---

- **Avoid long garbage collection (GC)**
  - This can result in soft failures of the Broker
- **GC tuning:**
  - `-XX:PermSize=96m -XX:MaxPermSize=96m -XX:+UseG1GC`
  - `-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35`
- **Enable GC logging:**
  - `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps`
  - `-XX:+PrintGCDetails -XX:+PrintTenuringDistribution`
  - `-XX:+PrintGCDetails -XX:+PrintGCDateStamps`
  - `-XX:+PrintTenuringDistribution -Xloggc:logs/gc.log`

# Kafka REST Proxy and Schema Registry

---

- **Confluent Platform includes a REST Proxy and Schema Registry**
- **REST Proxy:**
  - Allows applications written in languages other than Java to access the cluster
  - Needs CPU and memory
- **Schema Registry**
  - Allows Avro data to be stored in the cluster efficiently by storing schemas centrally (in a Kafka topic) rather than writing them along with each record
  - Needs few resources
- **Run multiple instances of each for high availability**
  - Use a VIP

# Kafka Configurations

---

- Most defaults don't need to be changed
- Check the 'Configurations' section of the Kafka documentation
  - <http://kafka.apache.org/documentation.html#configuration>

# Choosing The Number of Partitions (1)

---

- More partitions → higher throughput
  - $t$ : target throughput,  $p$ : Producer throughput per partition,  $c$ : Consumer throughput per partition  $\max(t/p, t/c)$
- Downside of increasing the number of partitions:
  - Requires more open filehandles
  - May increase unavailability
  - May increase end-to-end latency
  - May require more memory in the client
- Rule of thumb for maximums:
  - 2,000 to 4,000 partitions per Broker
  - Tens of thousands of partitions per cluster

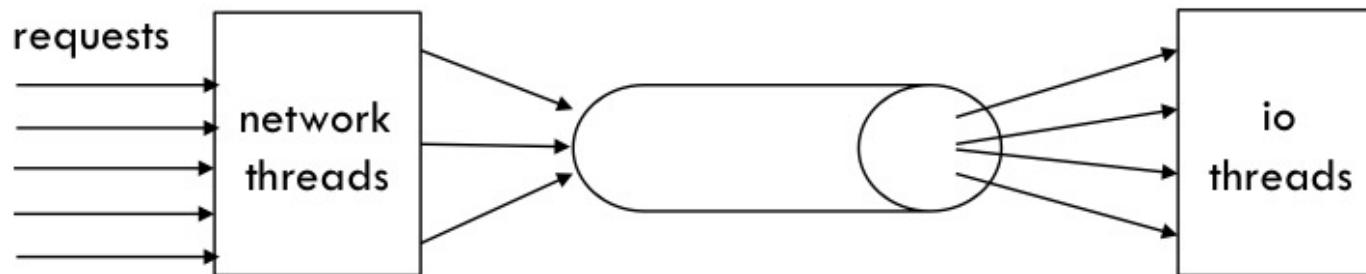
# Choosing The Number of Partitions (2)

---

- Be most careful if you are using keys
- Do not use more partitions than you need
  - Think of the number of threads initially; expand from there

# Configuring Worker Threads

---



- **num.network.threads (Increase for SSL)**
- **num.io.threads**
- **queued.max.requests**
  - Match with the number of clients per Broker

# Topic-Level Configuration Parameters

---

- **Retention**
  - retention.bytes
  - retention.ms
  - cleanup.policy
- **Rolling**
  - segment.bytes
  - segment.ms
- **Durability**
  - unclean.leader.election.enable
    - Can a new leader be elected from a non-in-sync replica?
  - min.insync.replicas
    - Minimum number of in-sync replicas required when committing a message

# Controlled Shutdown

---

- During a controlled Broker shutdown, reduce the unavailability window by moving leaders off the Broker before shutdown
  - `controlled.shutdown.enable`
  - `controlled.shutdown.max.retries`
  - `controlled.shutdown.retry.backoff.ms`

# Balancing the Leaders

---

- **Ideally, leaders should be evenly distributed among the Brokers**
  - Leaders do more work
  - Leaders can change on failure
- **Configuration options:**
  - `auto.leader.rebalance.enable`
  - `leader.imbalance.per.broker.percentage`
    - The ratio of leader imbalance allowed per Broker
  - `leader.imbalance.check.interval.seconds`
    - The frequency with which the partition rebalance check is triggered by the Controller

# Improving Startup Time

---

- **Startup on hard failure requires log recovery**
  - This is I/O- and CPU-intensive
  - Can take a long time, especially if there are many partitions
- **num.recovery.threads.per.data.dir**
  - Need to tune if using RAID

# Protecting Against Client Connection Leaks

---

- **max.connections.per.ip**
  - The maximum number of connections allowed to the Broker from a given IP address

# Message Size Limit

---

- Try not to change the maximum message size unless it is unavoidable
- Kafka is not optimized for very large messages
- If you must change the value, the following configuration parameters all need to be changed:
  - `message.max.bytes`
    - Global, or per-topic
  - `replica.fetch.max.bytes`
  - `max.partition.fetch.bytes`
    - Change this on *all* Consumers!

# Purgatory

---

- **Purgatory is part of the system used for stashing incomplete requests**
  - Produce requests waiting for acks from other replicas
  - Fetch requests waiting for more bytes
- **Completed requests need to be garbage-collected**
  - `fetch.purgatory.purge.interval.requests`
  - `producer.purgatory.purge.interval.requests`
    - If the number of garbage requests exceeds this value, clean the Fetch and Produce purgatory respectively

# Rack-Awareness (Kafka 0.10.x And Above)

---

- **Each Broker can be configured with a broker.rack property**
  - Examples: "rack-1", "us-east-1a"
- **Replicas will be placed on different racks**
  - Useful if deploying Kafka on Amazon EC2 instances across availability zones in the same region
  - Specify the same 'rack name' for brokers in the same availability zone
- **If not all brokers have broker.rack set:**
  - Automatic topic creation will ignore rack information
  - Manual topic creation will fail; use --ignore-racks to force creation (with no rack awareness)

# Runtime Configurations

---

- *Runtime Configuration Settings*
- **Chapter Review**

# Chapter Review

---

- Most Kafka defaults do not need to be changed
- Kafka is not optimized for large messages

# Monitoring and Alerting

Chapter 12



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

>>> 12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Monitoring and Alerting

---

- In this chapter you will learn:
  - Key metrics that you should consider monitoring

# Monitoring and Alerting

---

- ZooKeeper and OS-Level Monitoring
- Key Kafka Metrics
- Chapter Review

# Monitoring ZooKeeper

---

- **Use the RUOK command from ZooKeeper's command-line shell**
  - Responds with imok if it is running
- **Metrics to watch:**
  - Outstanding requests
  - Latency
  - Leader/Followers
  - Number of clients/watchers
- **Watch out for:**
  - Frequent leader elections
  - Number of znodes in the data tree
  - Amount of stored state
  - Number of open sessions
- **All ZooKeeper JMX metrics:**
  - <http://zookeeper.apache.org/doc/r3.4.6/zookeeperJMX.html>

# Monitoring Kafka at the OS Level

---

- **Items to watch:**

- Number of open file handles
    - Alert at 80% of the limit
  - Remaining disk space
    - Alert at 60% capacity
  - Network bytesIn/bytesOut
    - Alert at 60% capacity

# Monitoring and Alerting

---

- *ZooKeeper and OS-Level Monitoring*
- **Key Kafka Metrics**
- *Chapter Review*

# Kafka Metrics Format

---

- **Gauge**
  - Value
- **Meter**
  - OneMinuteRate, FiveMinuteRate, etc
  - Don't use count (this is the accumulated value since the Broker was started)
- **Histogram**
  - 50thPercentile, 95thPercentile, 99thPercentile, etc
- **Timer: Meter + Histogram**

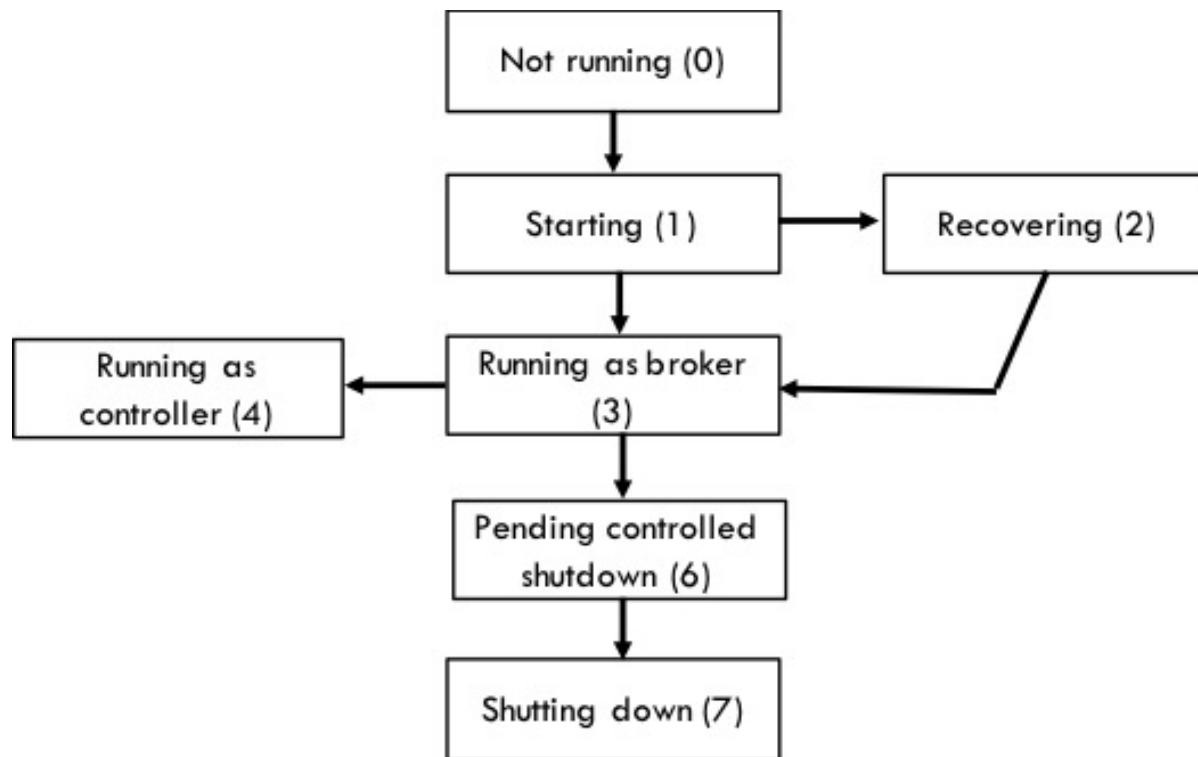
# Kafka Up and Running (1)

---

- All of these metrics are Gauge metrics
- Offline partition count
  - kafka.controller:type=KafkaController,name=OfflinePartitionsCount
  - Normal value is 0
- Under-replicated partition count
  - kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
  - Normal value is 0
- Who is the Controller?
  - kafka.controller:type=KafkaController,name=ActiveControllerCount
  - Only one Broker should have the value 1, the rest should have value 0

# Kafka Up and Running (2)

- This metric is a Gauge
- Kafka State
  - kafka.server:type=KafkaServer, name=BrokerState



# Throughput

---

- **All topics bytes/messages in (Meter)**
  - kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
  - kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
- **All topics bytes out (Meter)**
  - kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
- **Leader counts (Gauge)**
  - kafka.server:type=ReplicaManager,name=LeaderCount
- **Partition counts (Gauge)**
  - kafka.server:type=ReplicaManager,name=PartitionCount

# Latency

---

- All metrics are Histogram
- Produce/Fetch request time
  - kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Produce
  - kafka.network:type=RequestMetrics,name=TotalTimeMs,request=FetchConsumer
  - kafka.network:type=RequestMetrics,name=TotalTimeMs,request=FetchFollower
- Breakdown
  - kafka.network:type=RequestMetrics,name=LocalTimeMs,request=Produce
  - kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=Produce
  - kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=Produce

# Thread Capacity

---

- **I/O threads and network threads**

- kafka.server:type=KafkaRequestHandlerPool, name=RequestHandlerAvgIdlePercent
  - (Meter)
- kafka.network:type=SocketServer, name=NetworkProcessorAvgIdlePercent
  - (Gauge)
- Values are between 0 and 1

# Soft Failures

---

- **Soft failures can...**
  - Reduce availability
  - Expose corner case bugs
- **Typical causes**
  - Long Garbage Collection pauses
  - Network glitches
- **New metric in 0.10.x and above:**  
**kafka.server:type=SessionExpireListener, name=ZooKeeperExpiresPerSec**
  - Normal value is 0

# Other Useful JMX Metrics

---

- **Unclean leader election rate (Meter)**
  - kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec
- **Leader election rate (Meter)**
  - kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs
- **ISR expansion/shrink (Meter)**
  - kafka.server:type=ReplicaManager,name=IsrExpandsPerSec
  - kafka.server:type=ReplicaManager,name=IsrShrinksPerSec
    - Too frequent indicates a garbage collection issue
- **Maximum lag between Followers and Leaders (Gauge)**
  - kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica

# Monitoring and Alerting

---

- *ZooKeeper and OS-Level Monitoring*
- *Key Kafka Metrics*
- **Chapter Review**

# Chapter Review

---

- There are many metrics to monitor in order to ensure your cluster remains healthy
- Consider setting up alerting on key metrics

# Kafka Security

Chapter 13



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

>>> 13: Kafka Security

14: Kafka Streams



# Kafka Security

---

- In this chapter you will learn:
  - What security features Kafka 0.9 provides
  - How to configure encryption, authentication, and authorization on a cluster
  - How to migrate from an unsecure to a secure cluster

# Kafka Security

---

- **SSL for Encryption and Authentication**
- *SASL for Authentication*
- *Authorization*
- *Migration to a Secure Cluster*
- *Chapter Review*

# Encryption and Authentication Overview

---

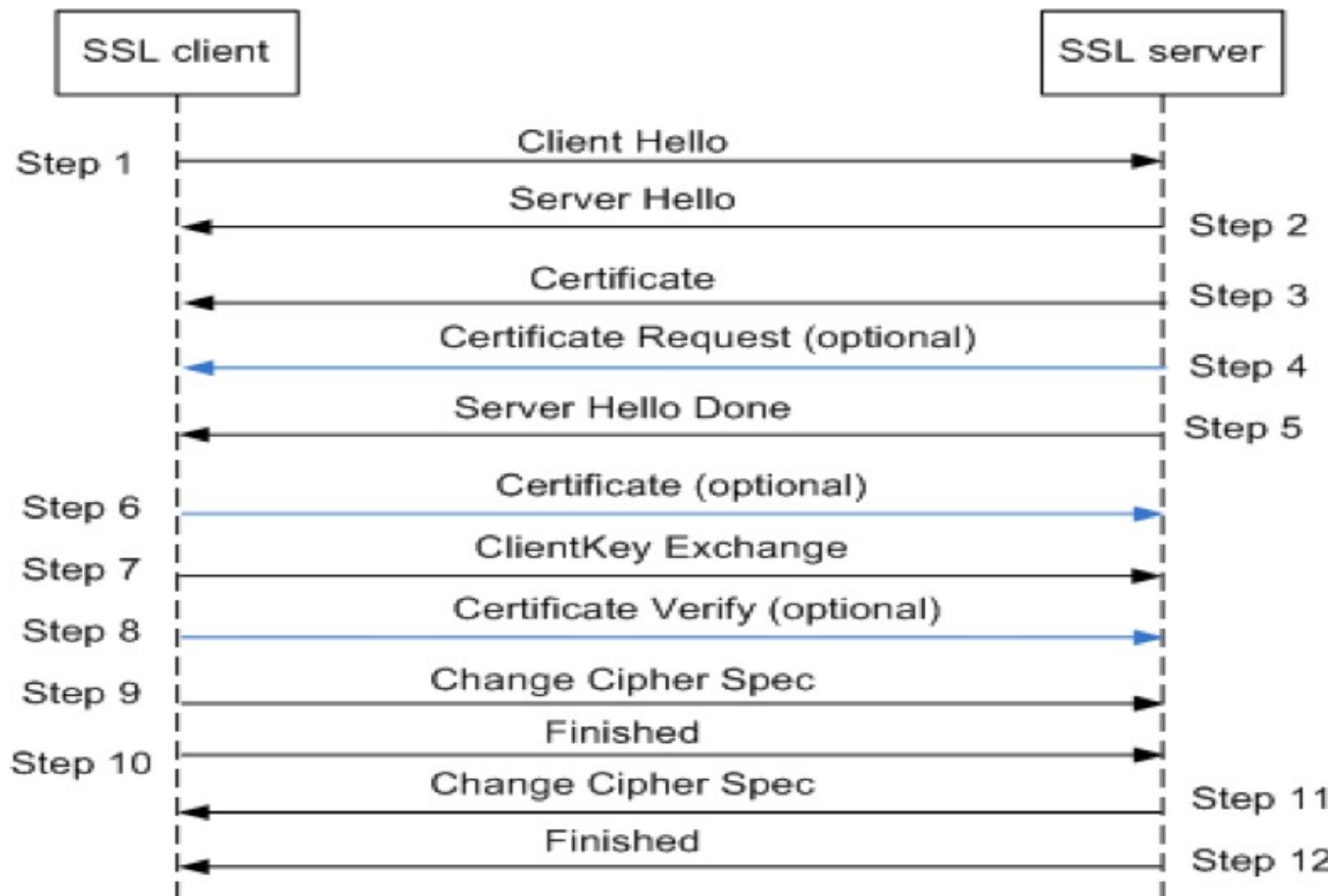
- **Authentication:** “Prove you are who you say you are”
- **Kafka Brokers support multiple ports**
  - Plain text (no wire encryption or authentication)
  - SSL (wire encryption and authentication)
  - SASL (Kerberos authentication)
  - SSL + SASL (SSL for wire encryption, SASL for authentication)
- **Clients choose which port to use**
  - They provide the required credentials through configurations
- **Note: Although the documentation refers to SSL, in fact the protocol used is TLS (Transport Layer Security)**

# Why is SSL Useful?

---

- **One-way authentication**
  - Secure wire transfer using encryption
- **Two-way authentication**
  - Broker knows the identity of the client
- **Easy to get started**
  - Just requires configuring the client and server
    - No extra servers needed

# The SSL Handshake



# Subsequent SSL Transfer

---

- After the handshake, data is encrypted with the agreed-upon cipher suite
- Note: there is an overhead involved with data encryption
  - Overhead to encrypt/decrypt the data
  - Can no longer use zero-copy data transfer in the Consumer

# 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%

# SSL Use-Cases

---

- **Wire encryption during cross-data center mirroring**
  - Just needs one-way authentication
- **For authorization based on SSL principals**
  - Needs two-way authentication

# Preparing SSL

---

1. Generate certificate (X509) in Broker key store
  2. Generate certificate authority (CA) for signing
  3. Sign Broker certificate with CA
  4. Import signed certificate and CA to Broker key store
  5. Import CA to client trust store
- Two-way authentication: Generate client certificate in a similar way

# Configuring SSL On The Cluster

- No client code change is needed
  - Just changes to the configuration files
- Client and Broker:

```
ssl.keystore.location = /var/private/ssl/kafka.server.keystore.jks  
ssl.keystore.password = test1234  
ssl.key.password = test1234  
ssl.truststore.location = /var/private/ssl/kafka.server.truststore.jks  
ssl.truststore.password = test1234
```

- Broker:

```
listeners = SSL://host.name:port  
security.inter.broker.protocol = SSL  
ssl.client.auth = required
```

- Client:

```
security.protocol = SSL
```

# 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

# Ensuring the Broker Host is Verified

---

- Set `ssl.endpoint.identification.algorithm` to `https`
  - The default is `null`, which does not verify the Broker host

# Kafka Security

---

- *SSL for Encryption and Authentication*
- **SASL for Authentication**
- *Authorization*
- *Migration to a Secure Cluster*
- *Chapter Review*

# What is SASL?

---

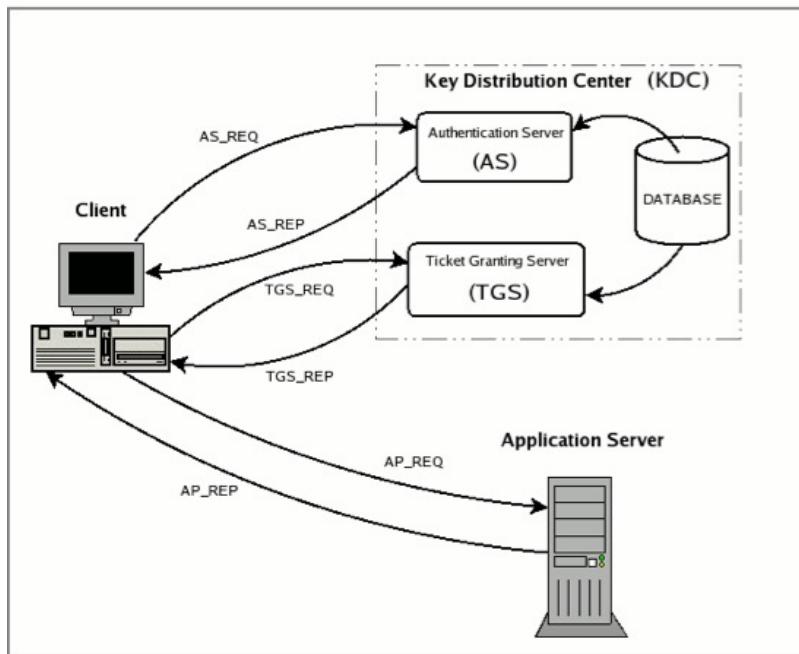
- **SASL: Simple Authentication and Security Layer**
  - Challenge/response protocols
  - Server issues challenge; client sends response
  - Continues until server is satisfied
- **SASL supports different mechanisms**
  - Plain: cleartext username/password (Kafka 0.10.x and above)
  - Digest MD5 (not supported yet)
  - GSSAPI: Kerberos (Kafka 0.9.x and above)

# Why Kerberos?

---

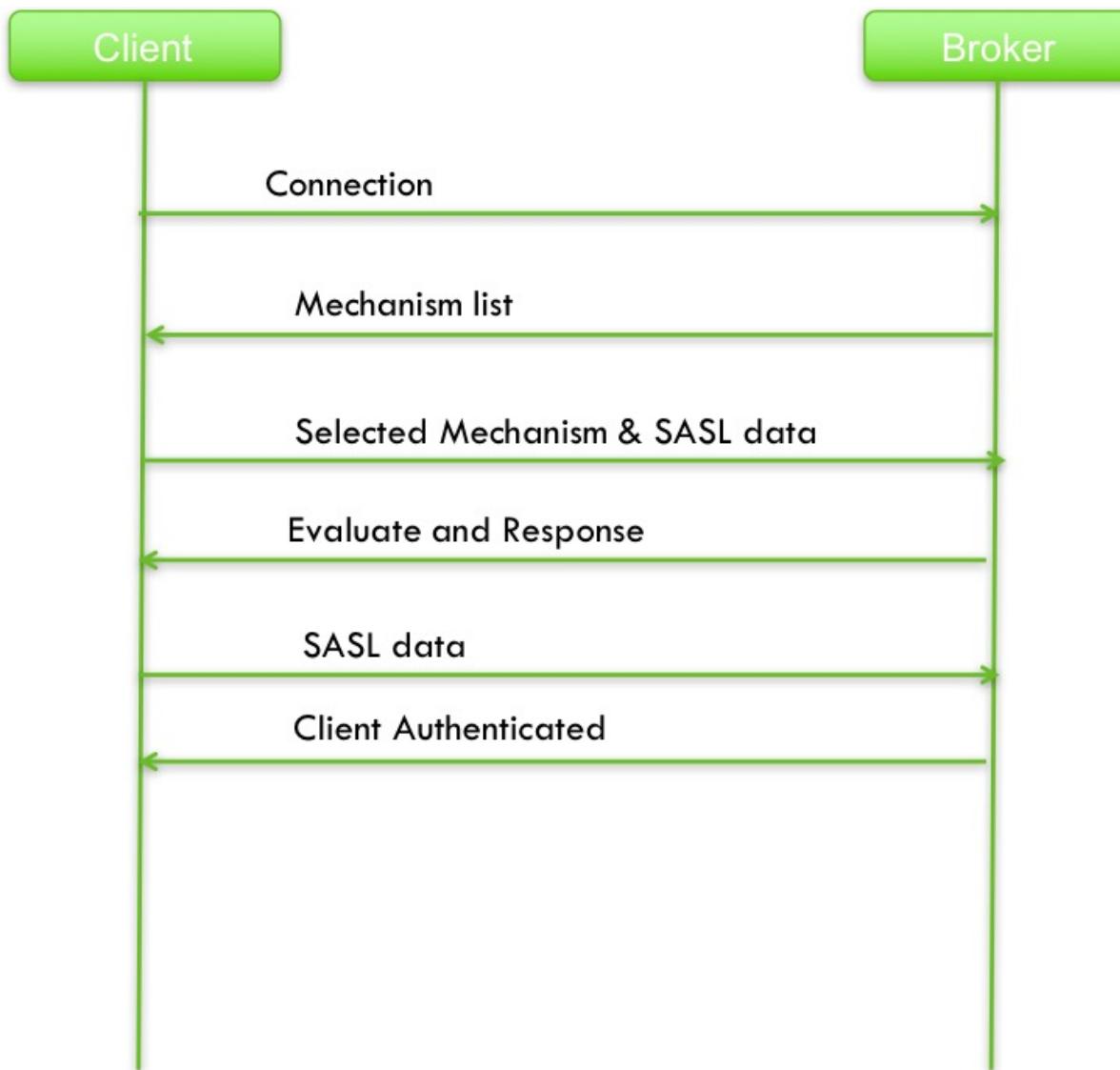
- **Kerberos provides secure single sign-on**
  - An organization may provide multiple services, but a user just needs a single Kerberos password to use all services
- **More convenient where there are many users**
- **Requires a Key Distribution Center (KDC)**
  - Each service and each user must register a Kerberos principal in the KDC

# How Kerberos Works



- Services authenticate with the KDC on startup
- Client authenticates with the AS on startup
- Client obtains a service ticket from TGS
- Client authenticates with the service using the service ticket

# SASL Handshake



# Data Transfer with SASL

---

- **SASL\_PLAINTEXT**
  - No wire encryption
- **SASL\_SSL**
  - Wire encryption using SSL

# Preparing Kerberos

---

- **Create the Kafka service principal in the KDC**
- **Create a Keytab for the Kafka principal**
  - Keytab includes the principal and encrypted Kerberos password
  - Allows authentication without typing a password
- **Create an application principal for the client KDC**
- **Create a Keytab for the application principal**

# Configuring Kerberos on the Broker

- Broker JAAS file

```
KafkaServer {  
    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";  
};
```

- Broker JVM

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

- Broker configuration file

```
listeners = SASL_PLAINTEXT://host.name:port (or SASL_SSL://host.name:port)  
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)  
sasl.kerberos.service.name=kafka
```

# Configuring Kerberos on the Client

- Client JAAS file

```
KafkaClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_client.keytab"  
    principal="kafka-client-1@EXAMPLE.COM";  
};
```

- Client JVM

```
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
```

- Client configuration file

```
security.protocol=SASL_PLAINTEXT(SASL_SSL)  
sasl.kerberos.service.name=kafka
```

# 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 username through sasl.kerberos.principal.to.local.rules**

# Kerberos Integration with Directory Service

---

- Can integrate with Active Directory
  - Need a new Authorization plugin for group integration

# Authentication Caveat

---

- **Authentication (SSL or SASL) happens once during socket connection**
  - No re-authentication occurs
- **If a certificate needs to be revoked, use authorization to remove the permission**

# SASL Caveat

---

- Default Kafka principal is host-specific: `kafka/kafka1.hostname.com@EXAMPLE.com`
- Client can only authenticate if connecting to the Broker host directly
  - This means you cannot use Virtual IP (VIP) addressing

# Configuring SASL PLAIN on the Broker

- Broker JAAS file

```
KafkaServer {  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
    // user and password for inter-broker communication  
    username="admin"  
    password="admin-secret"  
    // define user admin and a password  
    user_admin="admin-secret"  
    // define user alice and a password  
    user_alice="alice-secret"; }  
;
```

- Pass the JAAS file to the JVM

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

- Broker properties

```
listeners=SASL_SSL://host.name:port  
security.inter.broker.protocol=SASL_SSL  
sasl.mechanism.inter.broker.protocol=PLAIN  
sasl.enabled.mechanisms=PLAIN
```

# Configuring SASL PLAIN on the Client

- Client JAAS file

```
KafkaClient
{
    org.apache.kafka.common.security.plain.PlainLoginModule required
        username="alice" password="alice-secret";
}
;
```

- Pass the JAAS file to the JVM

```
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
```

- Client properties

```
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
```

# Enabling Multiple SASL Mechanisms on the Broker

---

- It is possible to enable multiple SASL mechanisms on the Broker
  - Useful for mixing internal (e.g., GSSAPI) and external (e.g., PLAIN) clients
  - Set `sasl.enabled.mechanisms` on the broker to GSSAPI , PLAIN
  - A client can only specify one SASL mechanism

# SASL PLAIN Caveat

---

- **The default implementation of SASL PLAIN on the Broker stores the username/password in a configuration file**
  - Need to make sure the file is protected
- **It is possible to plug in customized implementation if needed**
  - Details at <http://docs.confluent.io/3.0.0/kafka/sasl.html#use-of-sasl-plain-in-production>

# Kafka Security

---

- *SSL for Encryption and Authentication*
- *SASL for Authentication*
- **Authorization**
- *Migration to a Secure Cluster*
- *Chapter Review*

# Authorization Basics

---

- Authorization controls what permissions each authenticated principal has
- Authorization is pluggable, with a default implementation

# Access Control Lists (ACLs)

- Example: Alice is allowed to read data from Topic T1 from host Host1



# Principal

---

- Type + name
- Supported types: User
  - User:Alice
- Extensible, so users can add their own types (e.g., group)

# Operations and Resources

---

- **Operations:**
  - Read, Write, Create, Describe, ClusterAction, All
- **Resources:**
  - Topic, Cluster, and ConsumerGroup

Operations	Resources
<b>Read, Write, Describe (Read, Write implies Describe)</b>	Topic
<b>Read</b>	ConsumerGroup
<b>Create, ClusterAction</b> (communication between Controller and Brokers)	Cluster

# 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**

# Hosts

---

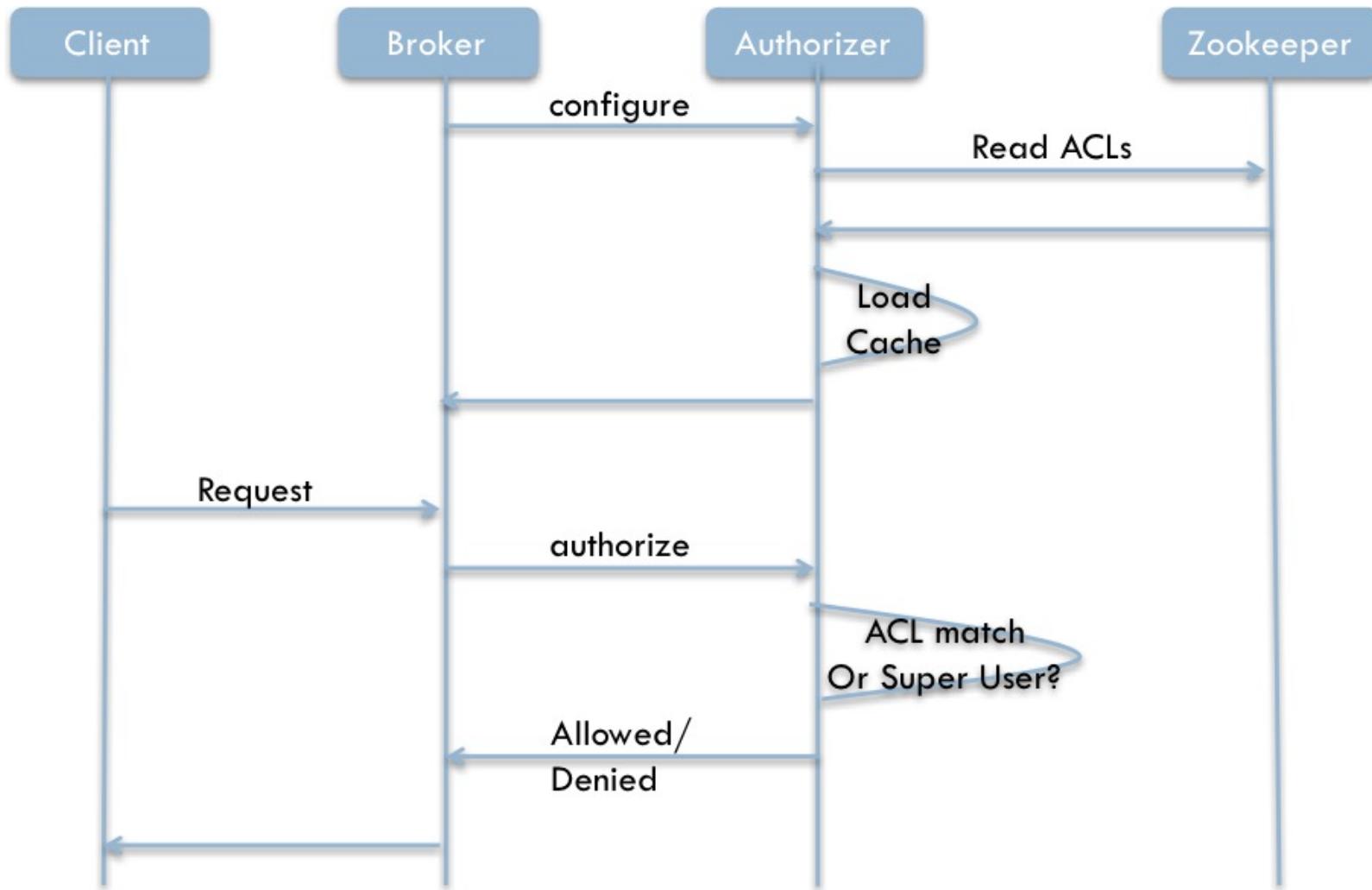
- Allows firewall-type security, even in a non-secure environment
  - Without needing system/network administrators to get involved

# SimpleAclAuthorizer

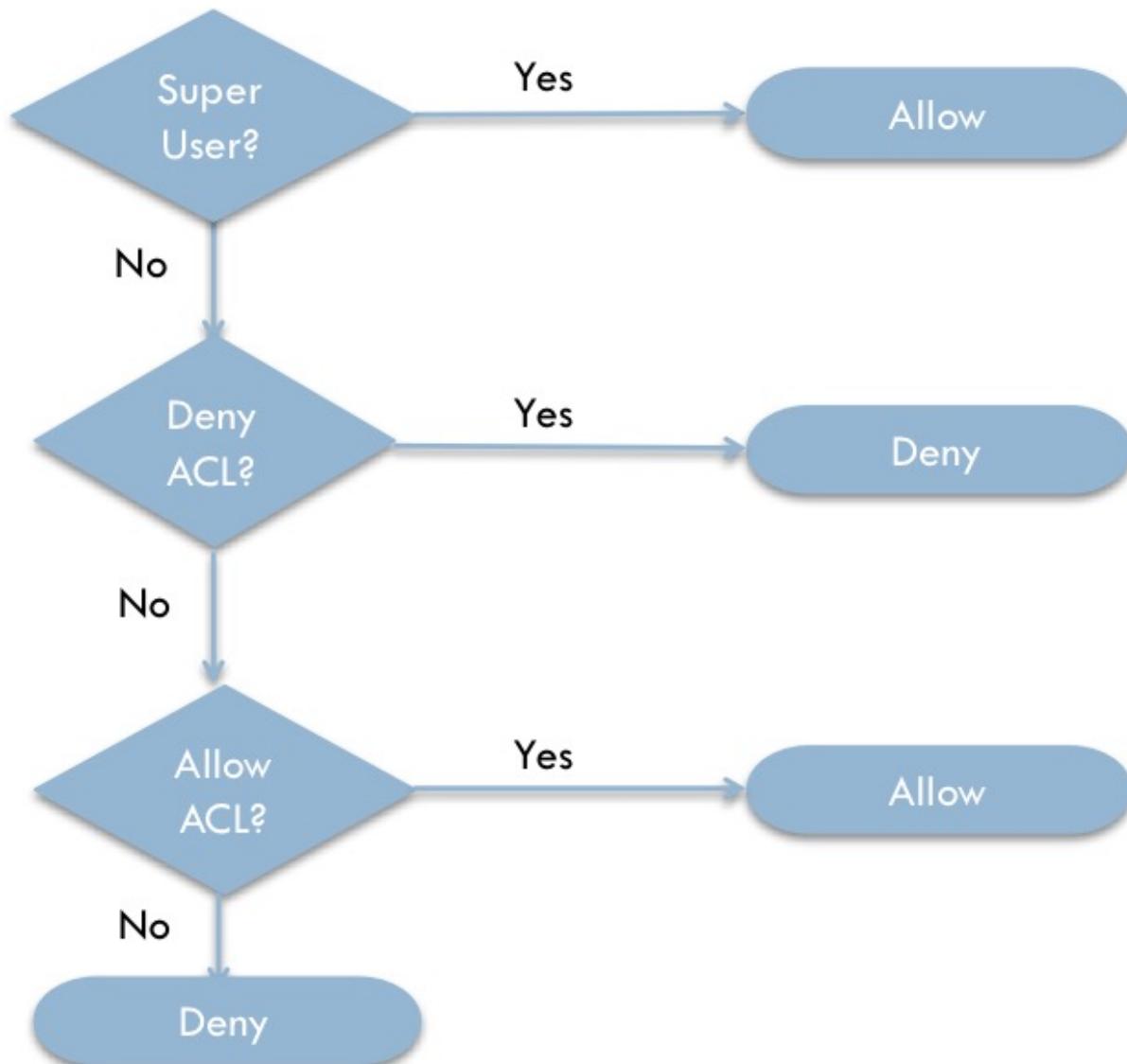
---

- **SimpleAclAuthorizer** is the default authorizer implementation
- Provides a CLI for adding and removing ACLs
- ACLs are stored in ZooKeeper and propagated to Brokers asynchronously
- ACLs are cached in the Broker for better performance

# Authorizer Flow



# Permission Check Sequence



# Configuring a Broker ACL

---

- `authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer`
- **Make Kafka principal super users**
  - Or grant ClusterAction and Read on all topics to the Kafka principal

# Configuring a Client ACL

- **Producer:**

- Grant Write on the Topic, Create on the cluster (for auto-creation)
- Or use --producer option in the CLI

```
bin/kafka-acls --authorizer-properties zookeeper.connect=zkhost:2181 \
--add --allow-principal User:Bob --producer --topic t1
```

- **Consumer:**

- Grant Read on the Topic, Read on the ConsumerGroup
- Or use the --consumer option in the CLI

```
bin/kafka-acls --authorizer-properties zookeeper.connect=zkhost:2181 \
--add --allow-principal User:Bob --consumer --topic t1 --group group1
```

# Authorizer Log

---

- The Authorizer log can be enabled through log4j
- Logs the decision on every request
  - Useful for debugging
  - Can serve as an audit log

# Wildcard Support

---

- Wildcard support is available for principal, resource, and host
- Use `--host=*` (instead of `--host *` ) from the CLI

# Securing ZooKeeper

---

- ZooKeeper stores:
  - Critical Kafka metadata
  - ACLs
- We need to prevent untrusted users from modifying this data

# ZooKeeper Security Integration

---

- ZooKeeper supports authentication through SASL
  - Kerberos or Digest MD5
- Set `zookeeper.set.acl` to true on every Broker
- Configure ZooKeeper user through the JAAS configuration file
- Each ZooKeeper path should be writable by its creator, readable by all

# ZooKeeper Client JAAS File

- ZooKeeper Client JAAS File:

```
Client
{
    com.sun.security.auth.module.Krb5LoginModule required
        useKeyTab=true
        storeKey=true
        keyTab="/etc/security/keytabs/kafka_server.keytab"
        principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};
```

# Kafka Security

---

- *SSL for Encryption and Authentication*
- *SASL for Authentication*
- *Authorization*
- **Migration to a Secure Cluster**
- *Chapter Review*

# Migrating From a Non-Secure to a Secure Kafka Cluster

---

- To migrate from a non-secure to a secure Kafka cluster:

- Configure Brokers with multiple ports

`listeners=PLAINTEXT://host.name:port,SSL://host.name:port`

- Gradually migrate clients to the secure port

- When done, turn off PLAINTEXT listener on all Brokers

# Migrating from a Non-Secure ZooKeeper to a Secure ZooKeeper

- To migrate from a non-secure ZooKeeper to a secure ZooKeeper:
  - Bounce every broker with a JAAS configuration file, with `zookeeper.set.acl false`
  - Bounce every broker again, with `zookeeper.set.acl true`
  - Run the following tool to set permissions on existing ZooKeeper paths:

```
kafka-run-class kafka.admin.ZkSecurityMigrator
```

# Kafka Security

---

- *SSL for Encryption and Authentication*
- *SASL for Authentication*
- *Authorization*
- *Migration to a Secure Cluster*
- **Chapter Review**

# Chapter Review

---

- **Kafka 0.9 supports both authentication and authorization**
  - Authentication via SSL or SASL/Kerberos
  - Authorization via ACLs (or other pluggable systems)
- **SSL provides encryption of data in flight, if required**
- **Authorization and authentication do not require code changes**
  - Just changes to configuration files

# Kafka Streams

Chapter 14



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

>>> 14: Kafka Streams



# Kafka Streams

---

- In this chapter you will learn:
  - The motivation behind Kafka Streams
  - The features provided by Kafka Streams
  - How to write an application using the Kafka Streams DSL (Domain-Specific Language)

# Kafka Streams

---

- An Introduction to Kafka Streams
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Chapter Summary*

# What Is Kafka Streams?

---

- **Kafka Streams is a lightweight Java library for building distributed stream processing applications using Kafka**
  - Easy to embed in your own applications
- **No external dependencies, other than Kafka**
- **Supports event-at-a-time processing (not microbatching) with millisecond latency**
- **Provides a table-like model for data**
- **Supports windowing operations, and stateful processing including distributed joins and aggregation**
- **Has fault-tolerance and supports distributed processing**
- **Includes both a Domain-Specific Language (DSL) and a low-level API**

# A Library, Not a Framework

---

- **Kafka Streams is an alternative to streaming frameworks such as**
  - Spark Streaming
  - Apache Storm
  - Apache Samza
  - etc.
- **Unlike these, it does not require its own cluster**
  - Can run on a stand-alone machine, or multiple machines

# Why Not Just Build Your Own?

---

- Many people are currently building their own stream processing applications
  - Using the Producer and Consumer APIs
- Using Kafka Streams is much easier than taking the ‘do it yourself’ approach
  - Well-designed, well-tested, robust
  - Means you can focus on the application logic, not the low-level plumbing

# Kafka Streams

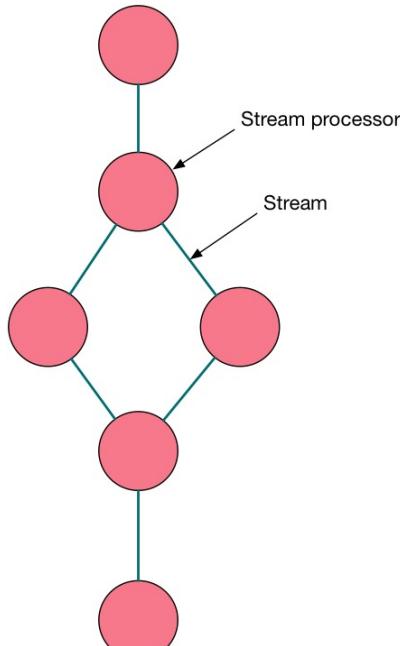
---

- *An Introduction to Kafka Streams*
- **Kafka Streams Concepts**
- *Creating a Kafka Streams Application*
- *Chapter Summary*

# Basic Concepts (1)

---

- A *stream* is an unbounded, continuously updating data set
- A *stream processor* transforms data in a stream
- A *processor topology* defines the data flow through the stream processors



Processor topology

# Basic Concepts (2)

---

- **Kafka Streams provides two ways to define a stream processor:**
  - The KafkaStreams DSL (Domain-Specific Language)
    - Provides the most common data transformation operations such as map, flatMap, countByKey,...
  - A low-level processor API
- **We will concentrate on the DSL**

# KStreams and KTables

---

- A KStream is an abstraction of a record stream
  - Each record represents a self-contained piece of data in the unbounded data set
- A KTable is an abstraction of a changelog stream
  - Each record represents an update
- Example: We send two records to the stream
  - ('apple', 1), and ('apple', 5)
- If we were to treat the stream as a KStream and sum up the values for apple, the result would be 6
- If we were to treat the stream as a KTable and sum up the values for apple, the result would be 5
  - The second record is treated as an update to the first, because they have the same key
- Typically, if you are going to treat a topic as a KTable it makes sense to configure log compaction on the topic

# Windowing, Joins, and Aggregations

---

- **Kafka Streams allows us to *window* the stream of data by time**
  - To divide it up into ‘time buckets’
- **We can join, or merge, two streams together based on their keys**
  - Typically we do this on windows of data
- **We can aggregate records**
  - Combine multiple input records together in some way into a single output record
  - Examples: sum, count
  - Again, this is usually done on a windowed basis

# Kafka Streams

---

- *An Introduction to Kafka Streams*
- *Kafka Streams Concepts*
- **Creating a Kafka Streams Application**
- *Chapter Summary*

# Configuring the Application

---

- Kafka Streams configuration is specified with a `StreamsConfig` instance
- This is typically created from a `java.util.Properties` instance
- Example:

```
Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-example");
streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "zookeeper1:2181");

// Continue to specify configuration options
streamsConfiguration.put(..., ...);
```

# Serializers and Deserializers (Serdes)

---

- Each Kafka Streams application must provide serdes (serializers and deserializers) for the data types of the keys and values of the records
  - These will be used by some of the operations which transform the data
- Set default serdes in the configuration of your application
  - These can be overridden in individual methods by specifying explicit serdes
- Configuration example:

```
streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().  
    getName());  
streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass().  
    getName());
```

# Available Serdes

- Kafka includes a variety of serdes in the `kafka-clients` Maven artifact

Data type	Serde
<code>byte[]</code>	<code>Serdes.ByteArray()</code> , <code>Serdes.Bytes()</code> ( <code>Bytes</code> wraps Java's <code>byte[]</code> and supports equality and ordering semantics)
<code>ByteBuffer</code>	<code>Serdes.ByteBuffer()</code>
<code>Double</code>	<code>Serdes.Double()</code>
<code>Integer</code>	<code>Serdes.Integer()</code>
<code>Long</code>	<code>Serdes.Long()</code>
<code>String</code>	<code>Serdes.String()</code>

- Confluent also provides `GenericAvroSerde` and `SpecificAvroSerde` as examples
  - See <https://github.com/confluentinc/examples/tree/kafka-0.10.0.0-cp-3.0.0/kafka-streams/src/main/java/io/confluent/examplesstreams/utils>

# Creating the Processing Topology

---

- Create a KStream or KTable object from one or more Kafka Topics using KStreamBuilder or KTableBuilder
  - For KTableBuilder only a single Topic can be specified
- Example:

```
KStreamBuilder builder = new KStreamBuilder();  
  
KStream<String, Long> purchases = builder.stream("PurchaseTopic");
```

# Transforming a Stream

---

- Data can be transformed using a number of different operators
- Some operations result in a new KStream object
  - For example, filter or map
- Some operations result in a KTable object
  - For example, an aggregation operation

# Some Stateless Transformation Operations (1)

---

- Examples of stateless transformation operations:
  - filter
    - Creates a new KStream containing only records from the previous KStream which meet some specified criteria
  - map
    - Creates a new KStream by transforming each element in the current stream into a different element in the new stream
  - mapValues
    - Creates a new KStream by transforming the value of each element in the current stream into a different element in the new stream

# Some Stateless Transformation Operations (2)

---

- Examples of stateless transformation operations (cont'd):
  - flatMap
    - Creates a new KStream by transforming each element in the current stream into zero or more different elements in the new stream
  - flatMapValues
    - Creates a new KStream by transforming the value of each element in the current stream into zero or more different elements in the new stream

# Some Stateful Transformation Operations

---

- **Examples of stateful transformation operations:**
  - countByKey
    - Counts the number of instances of each key in the stream; results in a new, ever-updating KTable
  - reduceByKey
    - Combines values of the stream using a supplied Reducer into a new, ever-updating KTable
- **For a full list of operations, see the JavaDocs at**  
**<http://docs.confluent.io/3.0.0streams/javadocs/index.html>**

# Writing Streams Back to Kafka

---

- Streams can be written to Kafka topics using the `to` method

- Example:

```
myNewStream.to("NewTopic");
```

- We often want to write to a topic but then continue to process the data

- Do this using the `through` method

```
myNewStream.through("NewTopic").flatMap(...);
```

# Printing A Stream's Contents

---

- It is sometimes useful to be able to see what a stream contains
  - Especially when testing and debugging
- `print()` writes the contents of the stream to `System.out`

# Running the Application

---

- To start processing the stream, create a `KafkaStreams` object
  - Configure it with your `KStreamBuilder` or `KTableBuilder` and your configuration
- Then call the `start()` method
- Example:

```
KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
streams.start();
```

- If you need to stop the application, call `streams.close()`

# A Simple Kafka Streams Example (1)

```
1 public class SimpleStreamsExample {  
2  
3  
4     public static void main(String[] args) throws Exception {  
5         Properties streamsConfiguration = new Properties();  
6         // Give the Streams application a unique name. The name must be unique in the Kafka cluster  
7         streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-streams-example");  
8         streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");  
9         streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "zookeeper1:2181");  
10        streamsConfiguration.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
11        // Specify default (de)serializers for record keys and for record values.  
12        streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.ByteArray().getClass()  
() .getName());  
13        streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass()  
() .getName());
```

# A Simple Kafka Streams Example (2)

```
1 KStreamBuilder builder = new KStreamBuilder();
2
3 // Read the input Kafka topic into a KStream instance.
4 KStream<byte[], String> textLines = builder.stream("TextLinesTopic");
5
6 // Convert to upper case (:: is Java 8 syntax)
7 KStream<byte[], String> uppercasedWithMapValues = textLines.mapValues(String::toUpperCase);
8
9 // Write the results to a new Kafka topic called "UppercasedTextLinesTopic".
10 uppercasedWithMapValues.to("UppercasedTextLinesTopic");
11
12 KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
13 streams.start();
14 }
15 }
```

# A More Complex Kafka Streams Application (1)

```
1 public class WordCountLambdaExample {  
2     public static void main(String[] args) throws Exception {  
3         Properties streamsConfiguration = new Properties();  
4         // Application name must be unique in the Kafka cluster  
5         streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-lambda-example");  
6  
7         streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");  
8         streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "zookeeper1:2181");  
9         // Specify default (de)serializers for record keys and for record values.  
10        streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass()  
11            .getName());  
11        streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass()  
12            .getName());  
12  
13        KStreamBuilder builder = new KStreamBuilder();
```

# A More Complex Kafka Streams Application (2)

```
1 // Construct a KStream from the input topic "TextLinesTopic"
2 KStream<String, String> textLines = builder.stream("TextLinesTopic");
3
4 KStream<String, Long> wordCounts = textLines
5 .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
6 .map((key, word) -> new KeyValue<>(word, word))
7     // Count the occurrences of each word (record key).
8     // This will change the stream type from KStream<String, String> to
9     // KTable<String, Long> (word -> count).
10 .countByKey("Counts")
11     // Convert the KTable<String, Long> into a `KStream<String, Long>` .
12 .toStream();
13
14 // Write the `KStream<String, Long>` to the output topic.
15 wordCounts.to(Serdes.String(), Serdes.Long(), "WordsWithCountsTopic");
16
17 // Now that we have finished the definition of the processing topology we can actually run
18 // it via `start()`. The Streams application as a whole can be launched just like any
19 // normal Java application that has a `main()` method.
20 KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
21 streams.start();
22 }
23 }
```

# Converting Our Avro Example to Kafka Streams (1)

---

- In a previous Hands-On Exercise, you wrote an application to turn a text-based Topic into an Avro topic
- With Kafka Streams, this becomes very simple

# Converting Our Avro Example to Kafka Streams (2)

```
1 import solution.model.ShakespeareKey;
2 import solution.model.ShakespeareValue;
3
4 // In the configuration section, add these lines. You will also need SpecificAvroSerde,
5 // SpecificAvroSerializer, and SpecificAvroDeserializer from Confluent's GitHub repo,
6 // under examples/kafka-streams/src/main/java/io/confluent/examplesstreams/utils
7 streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, SpecificAvroSerde.class);
8 streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, SpecificAvroSerde.class)
streamsConfiguration.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://schemaregistry1:8081");
9 // Then...
10 KStreamBuilder builder = new KStreamBuilder();
11
12 KStream<String, String> textLines = builder.stream(Serdes.String(), Serdes.String(),
"shakespeare_topic");
13 KStream<ShakespeareKey, ShakespeareValue> converted = textLines
14 .map((k, line) -> new KeyValue<ShakespeareKey, ShakespeareValue>(getShakespeareKey(k),
getShakespeareLine(line))); ①
15
16 converted.to("streaming_shakespeare_output");
17
18 KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
19 streams.start();
```

① These are the methods you created in the Exercise

# Parallelizing Kafka Streams

---

- Kafka Streams allows you to run an application across multiple machines
- It handles dividing the work across the machines
- It provides fault tolerance
  - If a machine fails, the task it was working on is automatically restarted on one of the remaining instances
- Machines store state locally, but changes to local state are persisted to a Kafka topic so they can be replayed if necessary
  - (This is similar to the approach taken by Apache Samza)

# Processing Guarantees

---

- Kafka streams supports *at-least-once* processing
- With no failures, it will process data exactly once
- If a machine fails, it is possible that some records may be processed more than once
  - Whether this is acceptable or not depends on the use-case
- Exactly-once processing semantics will be supported in a future version of Kafka Streams

# More Kafka Streams Features

---

- We have only scratched the surface of Kafka Streams
- Check the documentation to learn about processing windowed tables, joining tables, joining streams and tables...

# Comparing Kafka Streams to Other Solutions

	Kafka Streams	Storm	Spark Streaming	Flink
<i>Integration into existing infrastructure</i>	Easy (library)	Difficult	Difficult	Difficult
<i>Ease of Deployment</i>	Easy, flexible	Difficult	Difficult	Difficult
<i>Ease of operation</i>	Easy	Difficult (cluster required)	Difficult (cluster required)	Difficult (cluster required)
<i>Infrastructure footprint</i>	Small	Large (cluster required)	Large (cluster required)	Large (cluster required)
<i>Delivery Semantics</i>	At least once	At least once	Exactly once	Exactly once
<i>Latency</i>	Milliseconds	Seconds	Milliseconds	Milliseconds
<i>Fault Tolerance</i>	Yes	Yes	Yes	Yes
<i>Elastic scaling</i>	Yes	No	Yes	No

# How To Get Kafka Streams

---

- **Kafka Streams is a part of Kafka 0.10 and above**
  - Can be downloaded as part of Confluent Platform 3.x, or from [kafka.apache.org](http://kafka.apache.org)

# Kafka Streams

---

- *An Introduction to Kafka Streams*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- **Chapter Summary**

# Chapter Summary

---

- **Kafka Streams provides a DSL for writing Kafka stream processing applications in Java**
  - It is a lightweight library
  - No external dependencies other than Kafka
- **No external cluster is required, unlike most stream processing frameworks**

# Conclusion

Chapter 15



# Course Contents

---

01: Introduction

02: Kafka Fundamentals

03: Kafka's Architecture

04: Intra-Cluster Replication

05: Log Retention and Compaction

06: Developing With Kafka

07: More Advanced Kafka Development

08: Schema Management In Kafka

09: Kafka Connect for Data Movement

10: Basic Kafka Administration

11: Runtime Configurations

12: Monitoring and Alerting

13: Kafka Security

14: Kafka Streams



# Conclusion

---

- **During this course, you have learned:**
  - The types of data which are appropriate for use with Kafka
  - The components which make up a Kafka cluster
  - Kafka features such as Brokers, Topics, Partitions, and Consumer Groups
  - How to write Producers to send data to Kafka
  - How to write Consumers to read data from Kafka
  - Common patterns for application development
  - How to integrate Kafka with external systems using Kafka Connect
  - Kafka cluster administration and monitoring
  - How to write Kafka Streams applications

# Thank You!

---

- Thank you for attending the course
- Please complete the course survey (your instructor will give you details on how to access the survey)
- If you have any further feedback, please email [training-admin@confluent.io](mailto:training-admin@confluent.io)