

Kafka  
+  
Java

# Kafka Development Training

Content Credit: Confluent Team Source Presentation  
Internet, Kafka Documentation

# **Introduction**

# Course Contents

---

**>>> 01: Introduction**

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# Course Objectives

---

- **During this course, you will learn:**
  - The motivation for Apache Kafka
  - 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
  - How the REST Proxy supports development in languages other than Java
  - Common patterns for application development
  - How to integrate Kafka with external systems using Kafka Connect
  - Basic Kafka cluster administration
  - How to write streaming applications with Kafka Streams API and KSQL
- **Throughout the course, Hands-On Exercises will reinforce the topics being discussed**

**Hands-on using Java Language. Basic knowledge of Java is preferred.**

# The Motivation for Apache Kafka

# Course Contents

---

01: Introduction

**>>> 02: The Motivation for Apache Kafka**

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# The Motivation for Apache Kafka

---

- In this chapter you will learn:

- Some of the problems encountered when multiple complex systems must be integrated
- How processing stream data is preferable to batch processing
- The key features provided by Apache Kafka

# The Motivation for Apache Kafka

---

- **Systems Complexity**
- *Real-Time Processing is Becoming Prevalent*
- *Kafka: A Streaming Platform*
- *About Confluent*
- *Chapter Review*

# Simple Data Pipelines

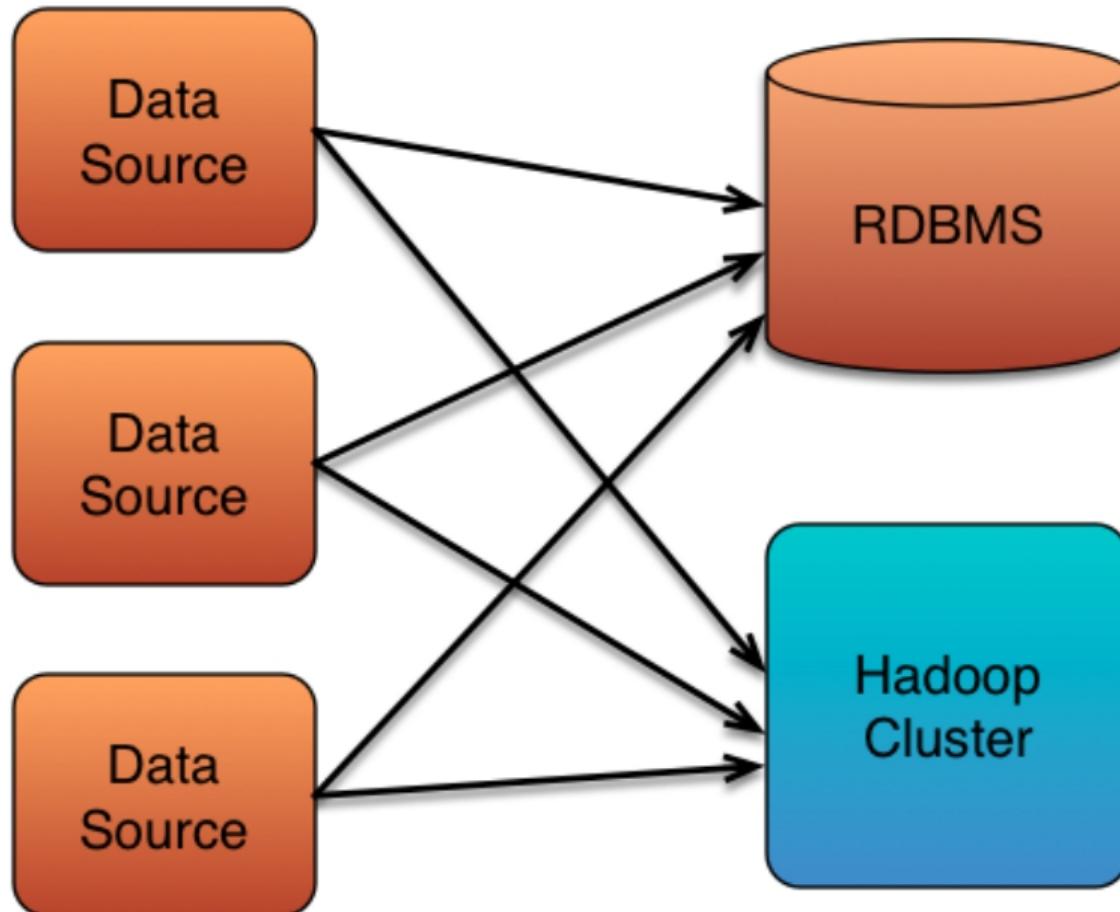
---

- **Data pipelines typically start out simply**
  - A single place where all data resides
  - A single ETL (Extract, Transform, Load) process to move data to that location
- **Data pipelines inevitably grow over time**
  - New systems are added
  - Each new system requires its own ETL procedures
- **Systems and ETL become increasingly hard to manage**
  - Codebase grows
  - Data storage formats diverge

# Small Numbers of Systems are Easy to Integrate

---

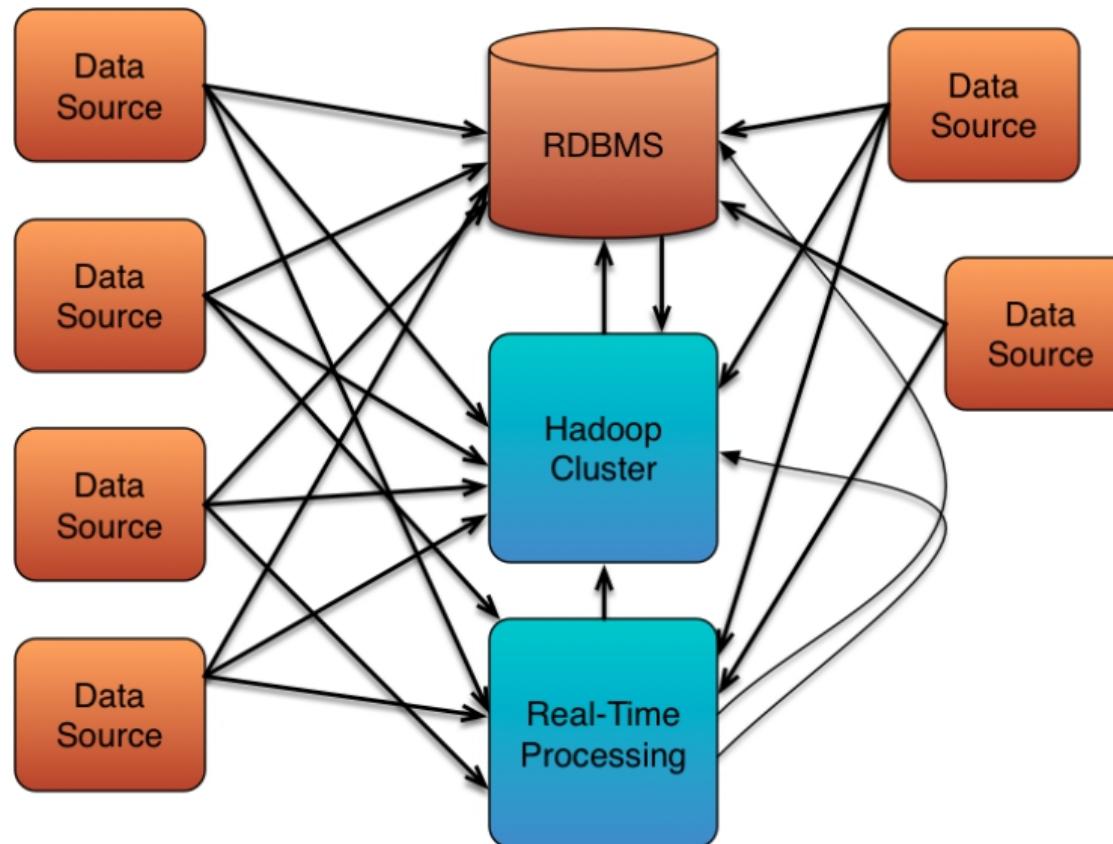
It is (relatively) easy to connect just a few systems together



# More Systems Rapidly Introduce Complexity (1)

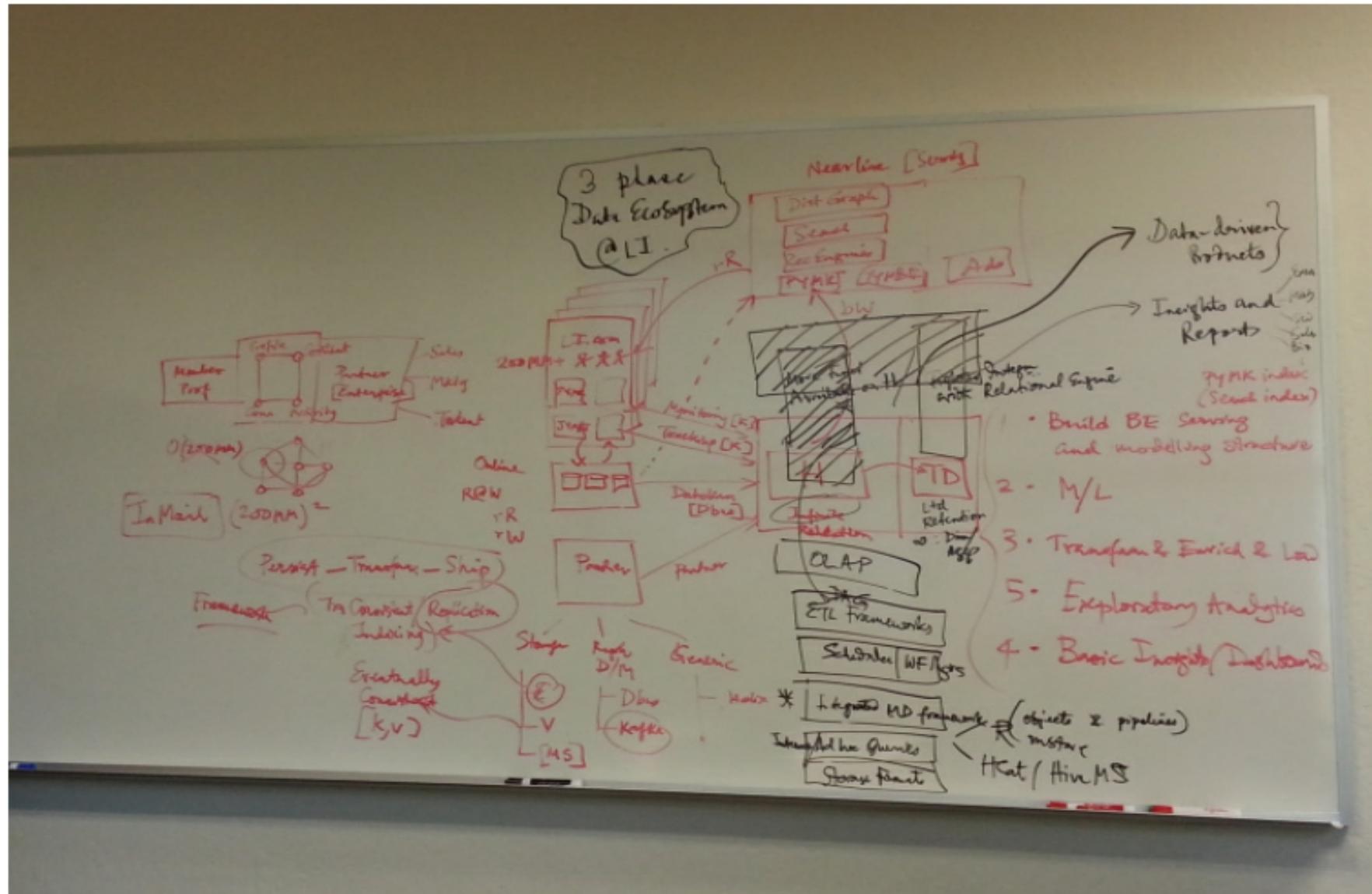
---

As we add more systems, complexity increases dramatically



# More Systems Rapidly Introduce Complexity (2)

...until eventually things become unmanageable



# The Motivation for Apache Kafka

---

- *Systems Complexity*
- **Real-Time Processing is Becoming Prevalent**
- *Kafka: A Streaming Platform*
- *About Confluent*
- *Chapter Review*

# Batch Processing: The Traditional Approach

---

- **Traditionally, almost all data processing was batch-oriented**
  - Daily, weekly, monthly...
- **This is inherently limiting**
  - “I can’t start to analyze today’s data until the overnight ingest process has run”

# Real-Time Processing: Often a Better Approach

---

- **These days, it is often beneficial to process data as it is being generated**
  - Real-time event processing allows real-time decisions
- **Use Cases:**
  - Fraud detection
  - Recommender systems for e-commerce web sites
  - Log monitoring and fault diagnosis
  - etc.
- **Of course, many legacy systems still rely on batch processing**
  - However, this is changing over time, as more ‘stream processing’ systems emerge
    - Kafka Streams
    - Apache Spark Streaming
    - Apache Storm
    - Apache Samza
    - etc.

# The Motivation for Apache Kafka

---

- *Systems Complexity*
- *Real-Time Processing is Becoming Prevalent*
- **Kafka: A Streaming Platform**
- *About Confluent*
- *Chapter Review*

# Kafka's Origins

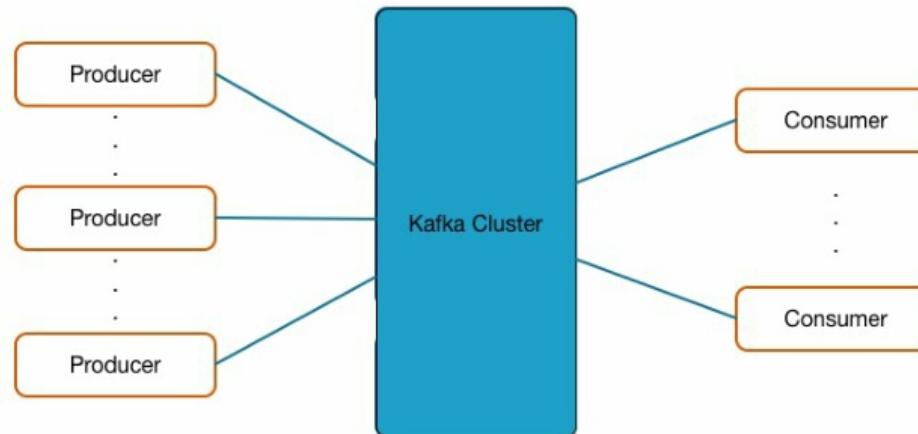
---

- **Kafka was designed to solve both problems**
  - Simplifying data pipelines
  - Handling streaming data
- **Originally created at LinkedIn in 2010**
  - Designed to support batch and real-time analytics
  - Kafka is now at the core of LinkedIn's architecture
  - Performs extremely well at very large scale
    - Produces over 2 *trillion* messages per day
- **An open source, top-level Apache project since 2012**
- **In use at many organizations**
  - Twitter, Netflix, Goldman Sachs, Hotels.com, IBM, Spotify, Uber, Square, Cisco...

# A Universal Pipeline for Data

---

- **Kafka decouples data source and destination systems**
  - Via a *publish/subscribe* architecture
- **All data sources write their data to the Kafka cluster**
- **All systems wishing to use the data read from Kafka**
- **Streaming platform**
  - Data integration: capture streams of events
  - Stream processing: continuous, real-time data processing and transformation



# The Motivation for Apache Kafka

---

- *Systems Complexity*
- *Real-Time Processing is Becoming Prevalent*
- *Kafka: A Streaming Platform*
- **About Confluent**
- *Chapter Review*

# About Confluent

---

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

# Confluent Open Source

---

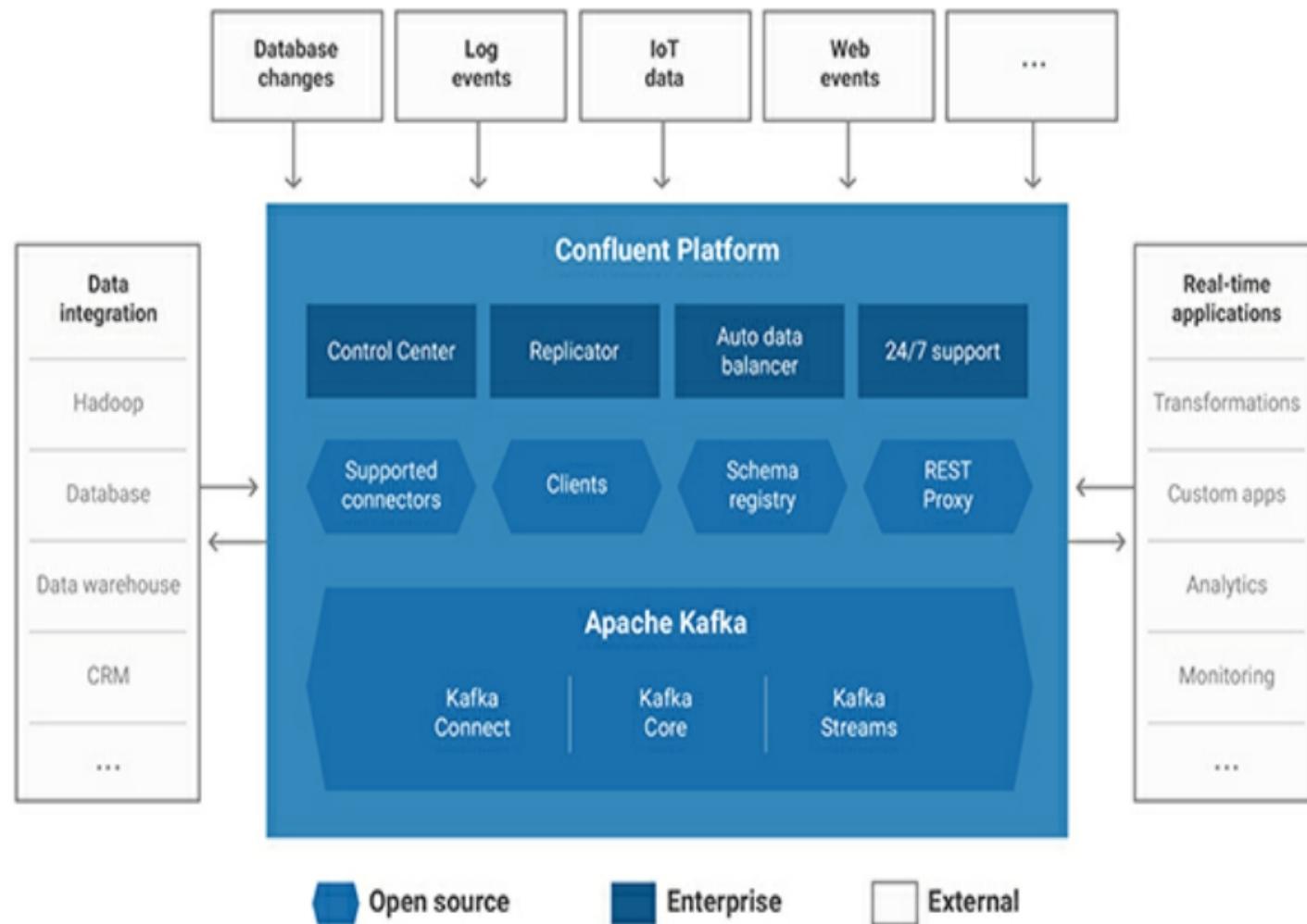
- **Confluent Open Source**
  - Completely free, open source
- **Easy to install**
  - Available as deb, RPM, Zip archive, tarball
  - Available in Docker
- **Provides Kafka plus extra ecosystem components**
  - Schema Registry
  - REST Proxy
  - Confluent CLI
  - Client libraries: Python, C/C++, Go, .NET
  - Additional Kafka Connect connectors

# Confluent Enterprise (1)

---

- **Confluent Enterprise**
  - Confluent Open Source, plus:
  - Confluent Control Center
    - Web-based management, end-to-end stream monitoring, alerting
    - Kafka Connect configuration
  - Replicator
    - Highly-available and durable multi-datacenter replication mechanism
    - Centralized configuration, management of multi-datacenter Kafka deployments
  - Auto Data Balancer
    - Shifts data to create an even workload across the cluster
    - Helps optimize resource utilization and cluster performance
  - Client library: Java Message Service (JMS)
  - Security plugin for the REST Proxy and Schema Registry
  - World-class enterprise support

# Confluent Enterprise (2)



# The Motivation for Apache Kafka

---

- *Systems Complexity*
- *Real-Time Processing is Becoming Prevalent*
- *Kafka: A Streaming Platform*
- *About Confluent*
- **Chapter Review**

# Chapter Review

---

- **Kafka was designed to simplify data pipelines, and to provide a way for systems to process streaming data**
- **Kafka can support batch and real-time analytics**
- **Kafka was started at LinkedIn and is now an open source Apache project with broad deployment**

# Kafka Fundamentals

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

**>>> 03: Kafka Fundamentals**

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# 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

# Kafka Fundamentals

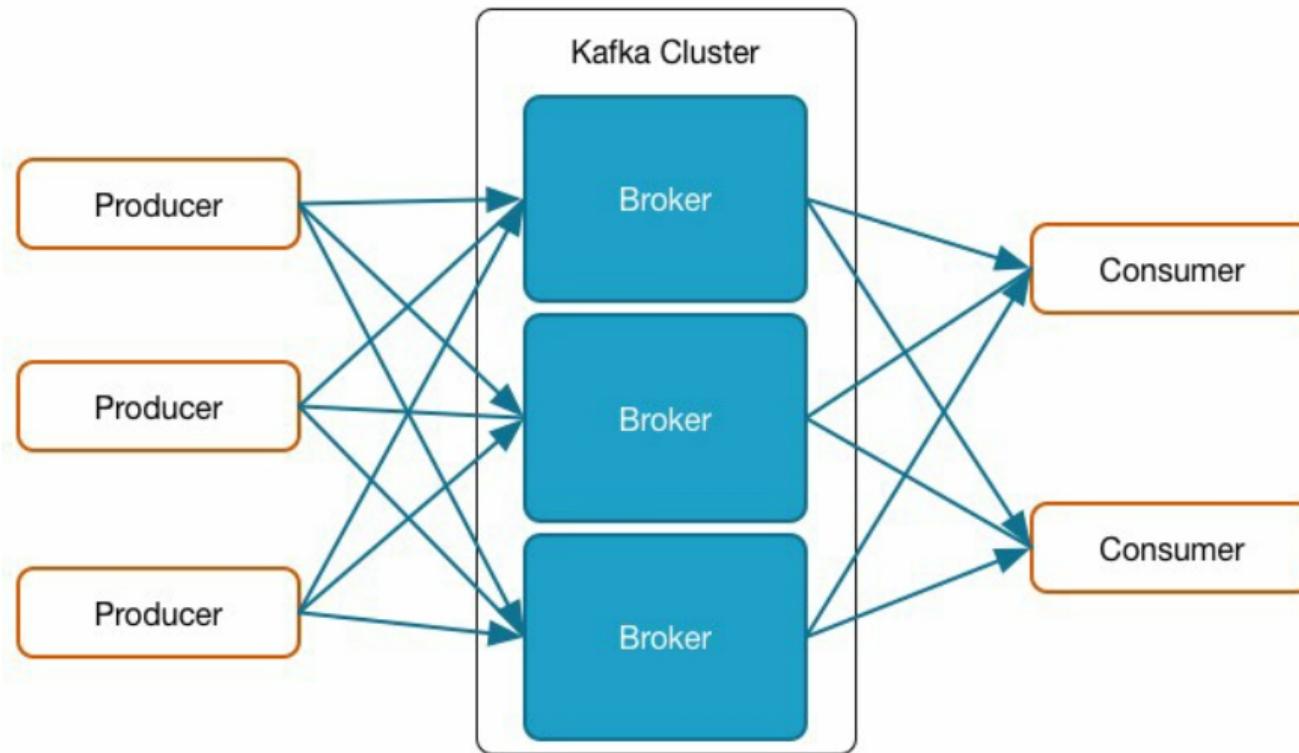
---

- **An Overview of Kafka**
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

# Reprise: A Very High-Level View of Kafka

---

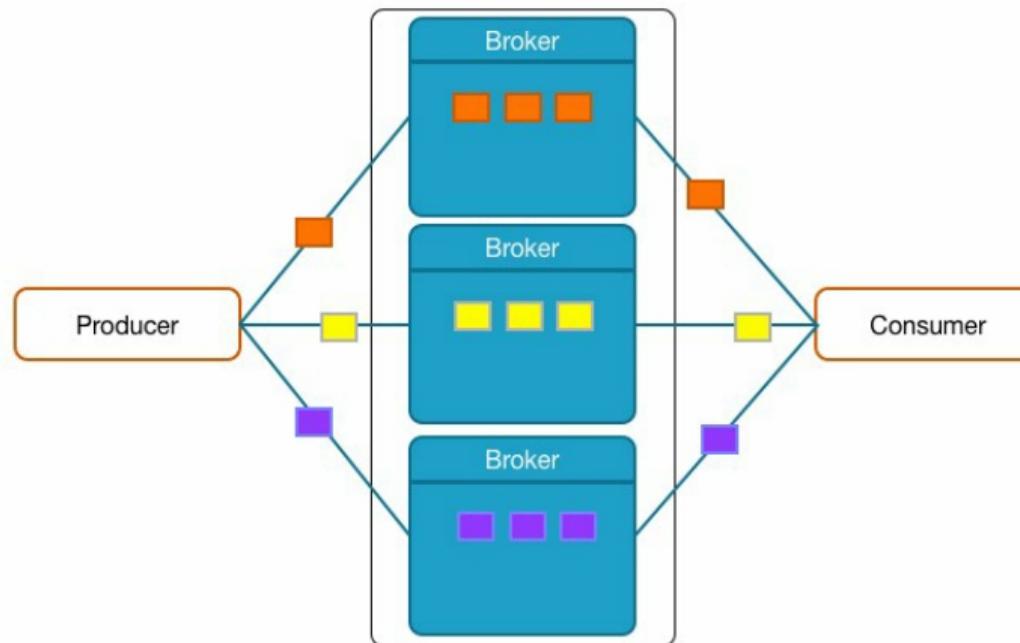
- **Producers** send data to the Kafka cluster
- **Consumers** read data from the Kafka cluster
- **Brokers** are the main storage and messaging components of the Kafka cluster



# Kafka Messages

---

- The basic unit of data in Kafka is a *message*
  - *Message* is sometimes used interchangeably with *record*
  - Producers write messages to Brokers
  - Consumers read messages from Brokers



# Key-Value Pairs

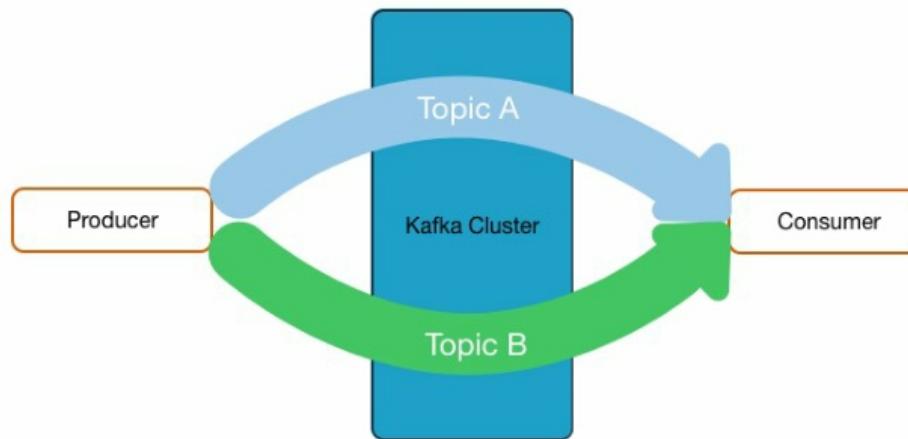
---

- **A message is a key-value pair**
  - All data is stored in Kafka as byte arrays
  - Producer provides serializers to convert the key and value to byte arrays
  - Key and value can be any data type

# Topics

---

- **Kafka maintains streams of messages called *Topics***
  - Logical representation
  - They categorize messages into groups

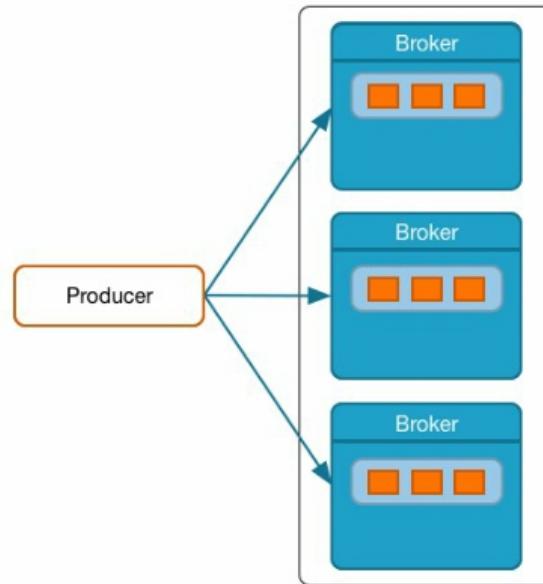


- **Developers decide which Topics exist**
  - By default, a Topic is auto-created when it is first used
- **One or more Producers can write to one or more Topics**
- **There is no limit to the number of Topics that can be used**

# Partitioned Data Ingestion

---

- Producers shard data over a set of *Partitions*
  - Each Partition contains a subset of the Topic's messages
  - Each Partition is an ordered, immutable log of messages
- Partitions are distributed across the Brokers
- Typically, the message key is used to determine which Partition a message is assigned to



# Load Balancing and Semantic Partitioning

---

- **Producers use a partitioning strategy to assign each message to a Partition**
- **Having a partitioning strategy serves two purposes**
  - Load balancing: shares the load across the Brokers
  - Semantic partitioning: user-specified key allows locality-sensitive message processing
- **The partitioning strategy is specified by the Producer**
  - Default strategy is a hash of the message key
    - $\text{hash}(\text{key}) \% \text{ number\_of\_partitions}$
  - If a key is not specified, messages are sent to Partitions on a round-robin basis
- **Developers can provide a custom partitioner class**

# 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*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

# Producer Basics

---

- Producers write data in the form of *messages* to the Kafka cluster
- Producers can be written in any language
  - Native Java, C/C++, Python, Go, .NET, JMS 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 Fundamentals

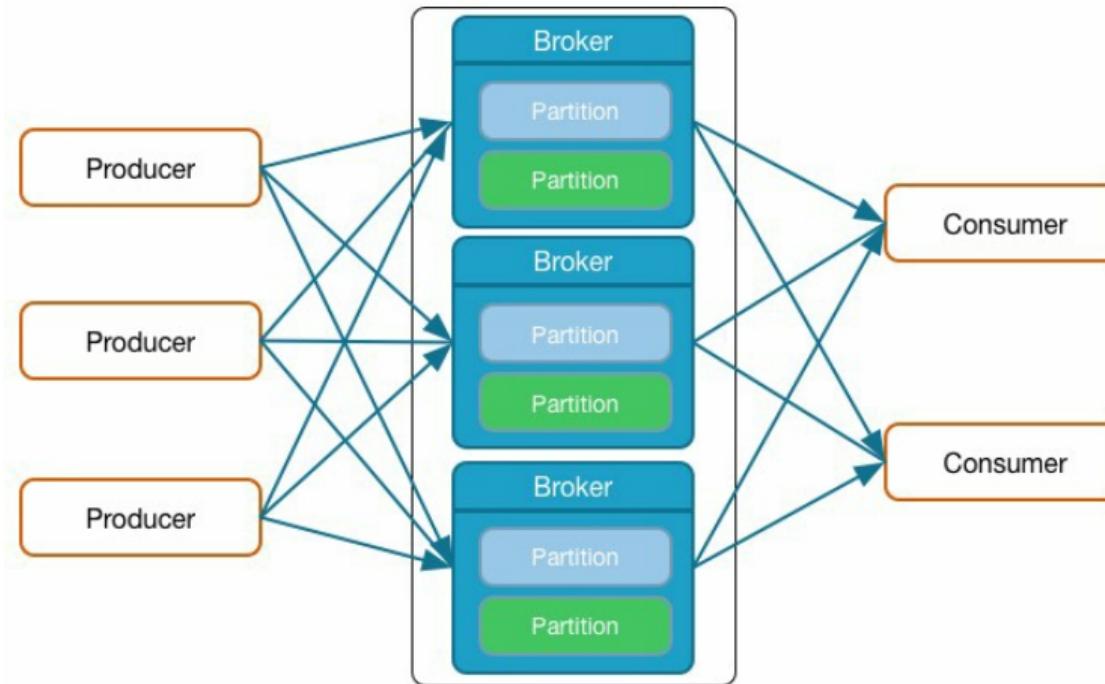
---

- *An Overview of Kafka*
- *Kafka Producers*
- **Kafka Brokers**
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

# Broker Basics

---

- Brokers receive and store messages when they are sent by the Producers
- A production Kafka cluster will have three or more Brokers
  - Each can handle hundreds of thousands, or millions, of messages per second



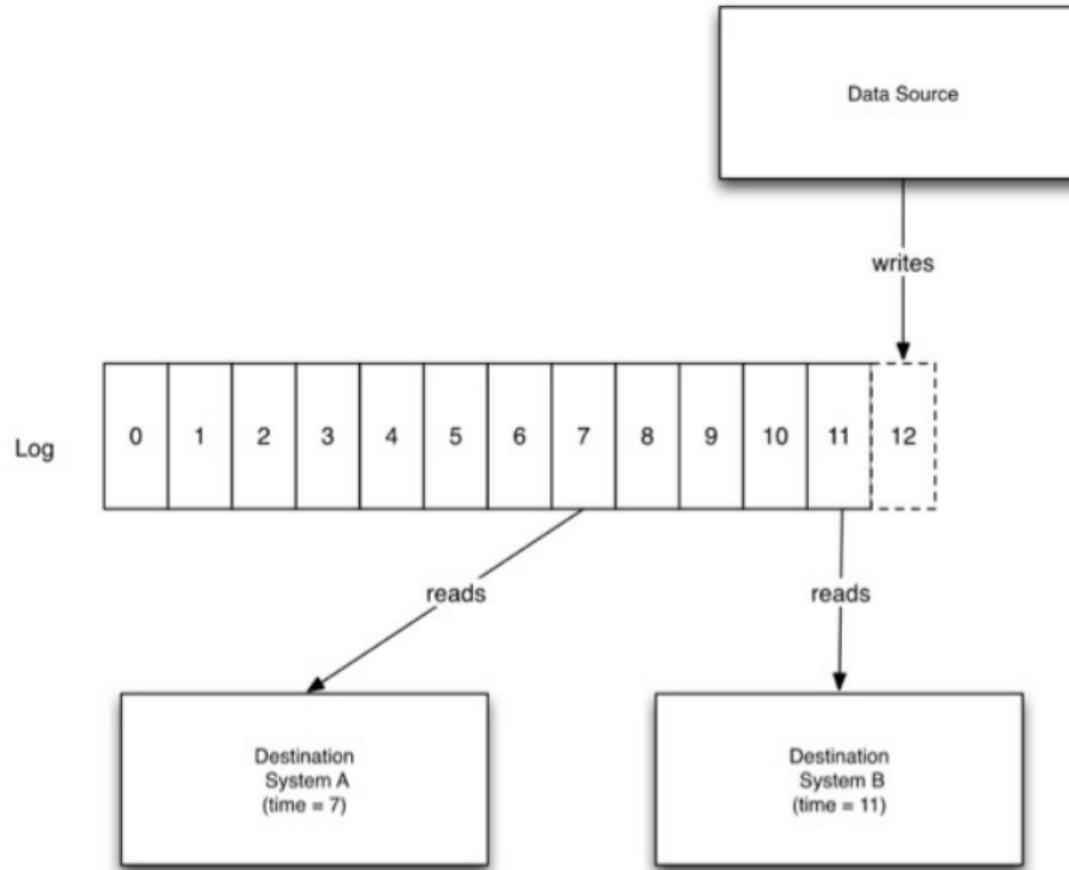
# Brokers Manage Partitions

---

- **Messages in a Topic are spread across Partitions in different Brokers**
- **Typically, a Broker manages multiple Partitions**
- **Each Partition is stored on the Broker's disk as one or more log files**
  - Not to be confused with log4j files used for monitoring
- **Each message in the log is identified by its *offset* number**
  - A monotonically increasing value
- **Kafka provides a configurable retention policy for messages to manage log file growth**
  - Retention policies can be configured per Topic

# Messages are Stored in a Persistent Log

---



# Metadata in a Kafka Message

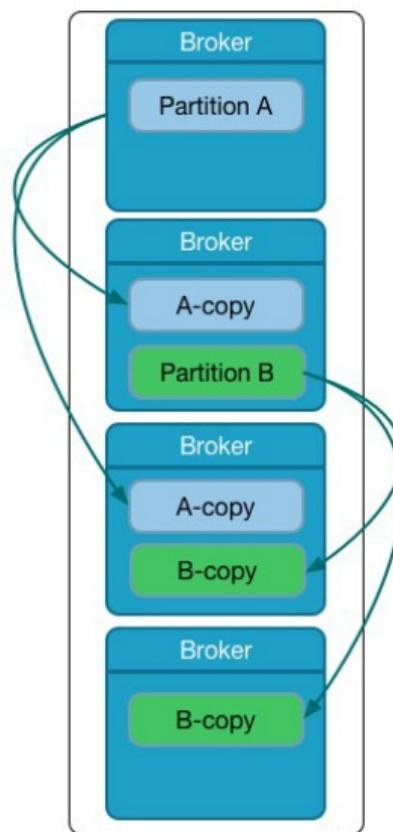
---

- **A Kafka Message contains data and metadata**
  - Key/value pair
  - Offset
  - Timestamp
  - Compression type
  - Magic byte
  - Optional message headers API
    - Application teams can add custom key-value paired metadata to messages
    - Additional fields to support batching, exactly once semantics, replication protocol
- **Latest message format:**  
<http://kafka.apache.org/documentation.html#messageformat>

# Fault Tolerance via a Replicated Log

---

- Partitions can be replicated across multiple Brokers
- Replication provides fault tolerance in case a Broker goes down
  - Kafka automatically handles the replication



# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- **Kafka Consumers**
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

# Consumer Basics

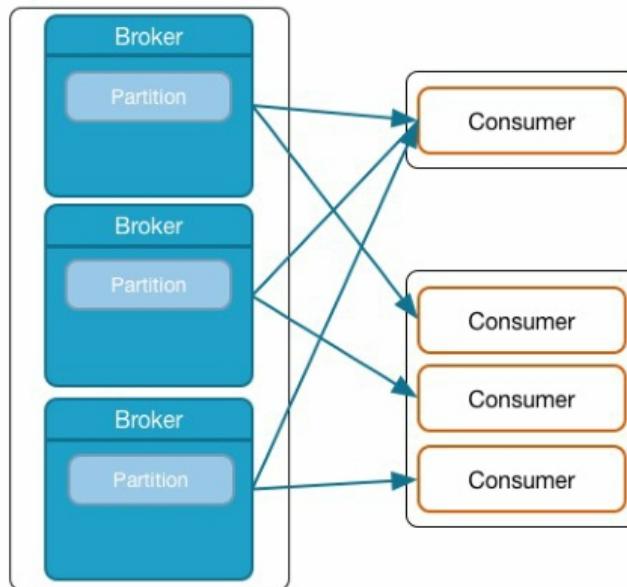
---

- **Consumers pull messages from one or more Topics in the cluster**
  - 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**
- **A command-line Consumer tool exists to read messages from the cluster**
  - Useful for testing, debugging, etc.

# Distributed Consumption

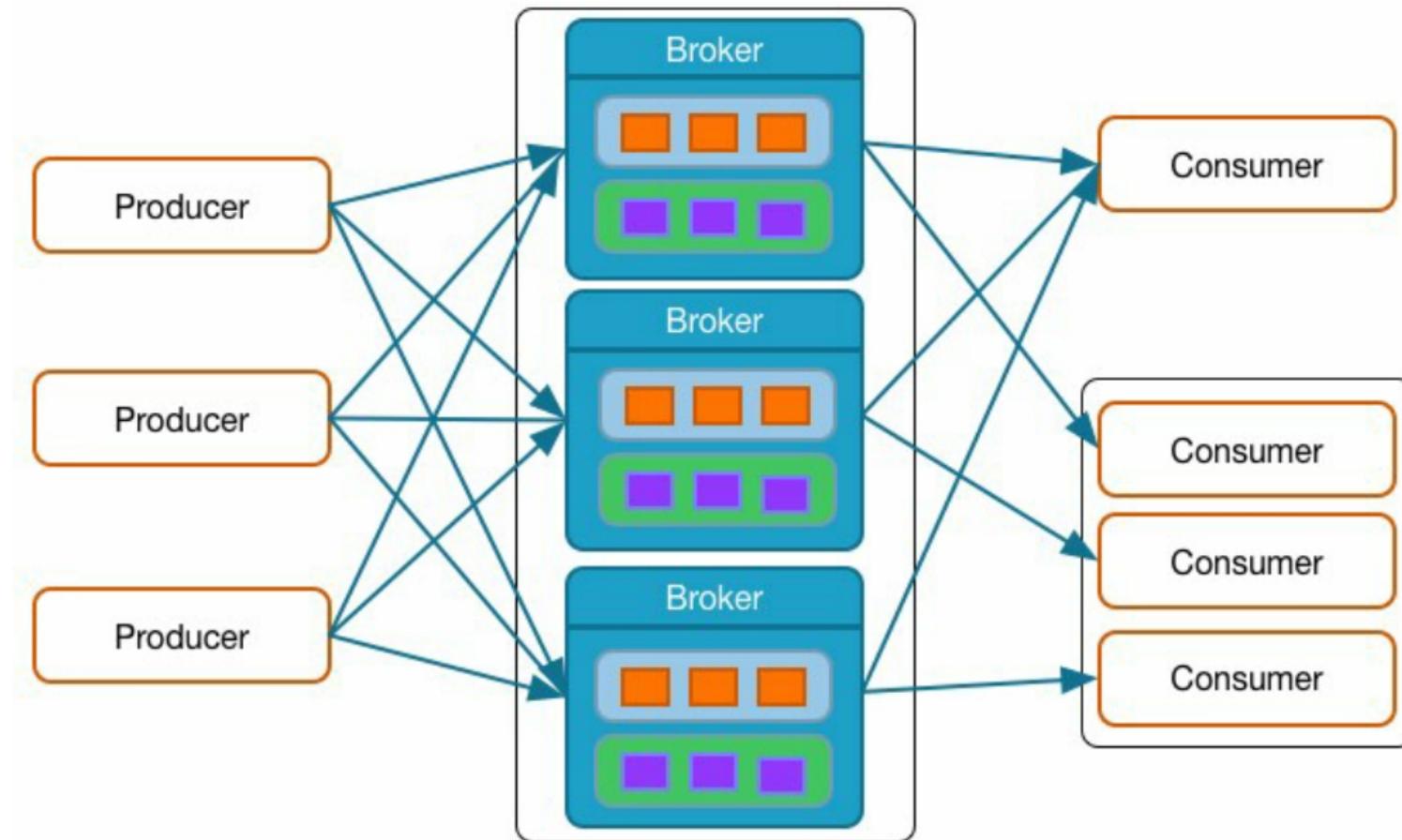
---

- **Different Consumers can read data from the same Topic**
  - By default, each Consumer will receive all the messages in the Topic
- **Multiple Consumers can be combined into a *Consumer Group***
  - Consumer Groups provide scaling capabilities
  - Each Consumer is assigned a subset of Partitions for consumption



# The Result: A Linearly Scalable Firehose

---



# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- **Kafka's Use of ZooKeeper**
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

# What is ZooKeeper?

---

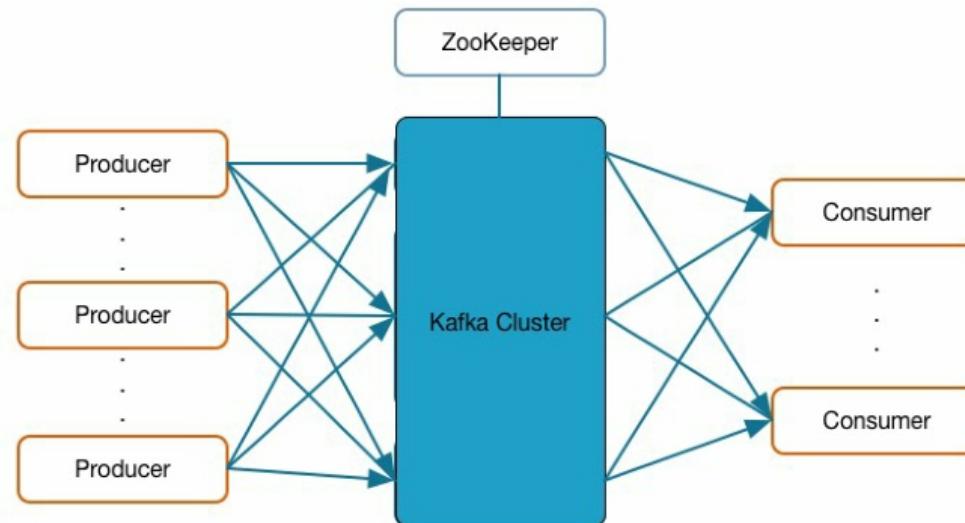
- **ZooKeeper is a centralized service that can be used by distributed applications**
  - Open source Apache project
  - Enables highly reliable distributed coordination
  - Maintains configuration information
  - Provides distributed synchronization
- **Used by many projects**
  - Including Kafka and Hadoop
- **Typically consists of three or five servers in an *ensemble***
  - This provides resiliency should a machine fail



# How Kafka Uses ZooKeeper

---

- **Kafka Brokers use ZooKeeper for a number of important internal features**
  - Cluster management
  - Failure detection and recovery
  - Access Control List (ACL) storage



# Quiz: Question

---

- **Provide the correct relationship — 1:1, 1:N, N:1, or N:N**
  - Broker to Partition — ?
  - Key to Partition — ?
  - Producer to Topic — ?
  - Consumer Group to Topic — ?
  - Consumer (in a Consumer Group) to Partition — ?

# Quiz: Answer

---

- **Provide the correct relationship — 1:1, 1:N, N:1, or N:N**
  - Broker to Partition — N:N
  - Key to Partition — N:1
  - Producer to Topic — N:N
  - Consumer Group to Topic — N:N
  - Consumer (in a Consumer Group) to Partition — 1:N

# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- **Comparisons with Traditional Message Queues**
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

# Kafka's Benefits over Traditional Message Queues

---

- **Durability and Availability**
  - Messages are replicated on multiple machines for reliability
  - Cluster can handle Broker failures
- **Excellent scalability**
  - Even a small cluster can process a large volume of messages
  - Tests have shown that three low-end machines can easily deal with two million writes per second
- **Very high throughput**
- **Supports both real-time and batch consumption**
- **Data retention**

# Multi-Subscription and Scalability

---

- **Multi-subscription provides easy distribution of data**
  - Once the data is in Kafka, it can be read by multiple different Consumers
  - For instance, a Consumer which writes the data to the Hadoop Distributed File System (HDFS), another to do real-time analysis on the data, etc.
- **Multiple Brokers, multiple Topics, and Consumer Groups provide very high scalability**
  - Kafka was designed as a distributed system from the ground up
  - Enables parallelism

# The Advantages of a Pull Architecture

---

- **Kafka Consumers pull messages from the Brokers**
  - 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
    - A slow Consumer will not affect Producers

# The Page Cache for High Performance

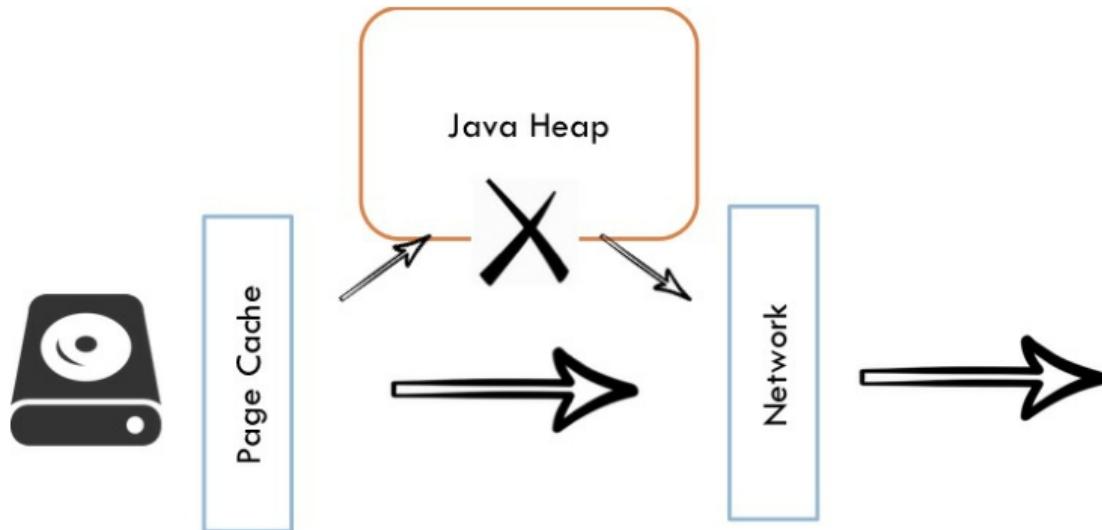
---

- **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

# Speeding Up Data Transfer

---

- Kafka uses zero-copy data transfer (Broker → Consumer)



# Kafka Fundamentals

---

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- **Hands-On Exercise: Using Kafka's Command-Line Tools**
- *Chapter Review*

# 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*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- **Chapter Review**

# Chapter Review

---

- **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 Architecture

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

>>> **04: Kafka's Architecture**

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# Kafka's Architecture

---

- How Kafka's log files are stored on the Kafka Brokers
- How Kafka uses replicas for reliability
- How Consumer Groups and Partitions provide scalability

# Kafka's Architecture

---

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

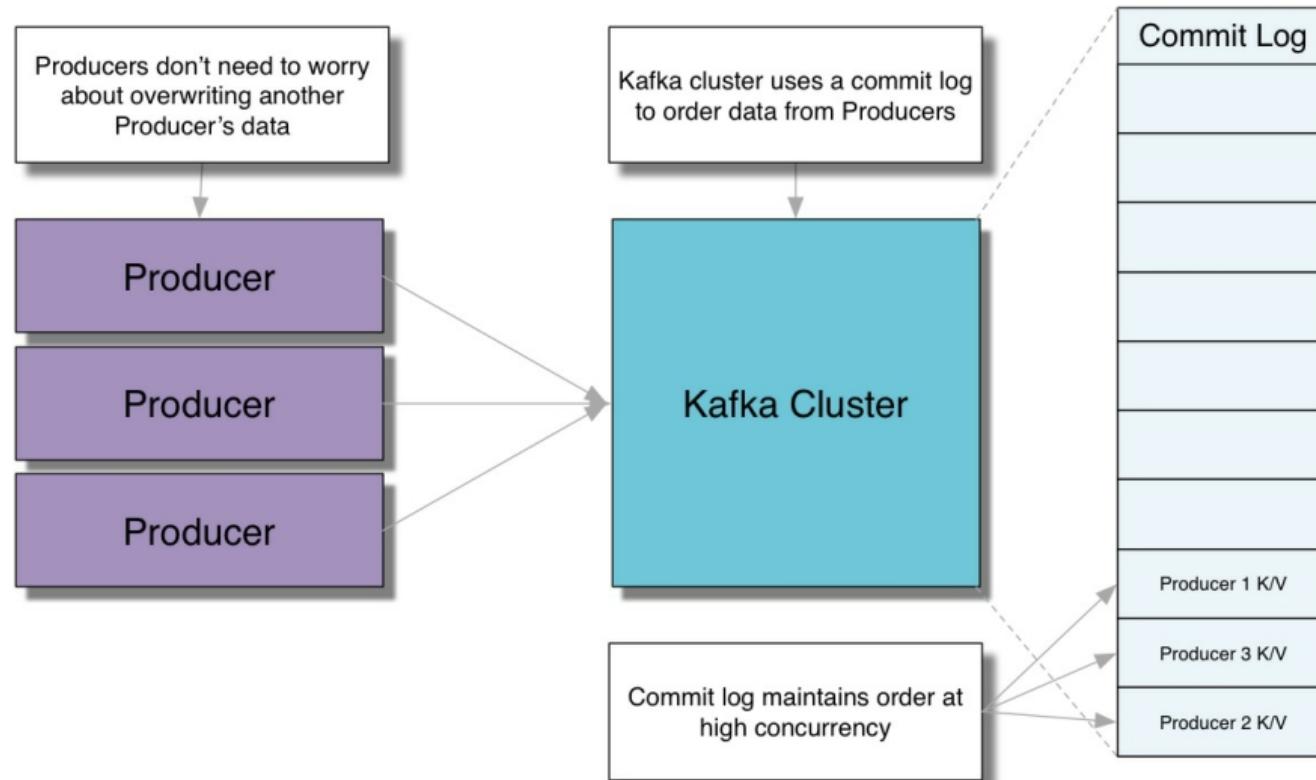
# What is a Commit Log?

---

- 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

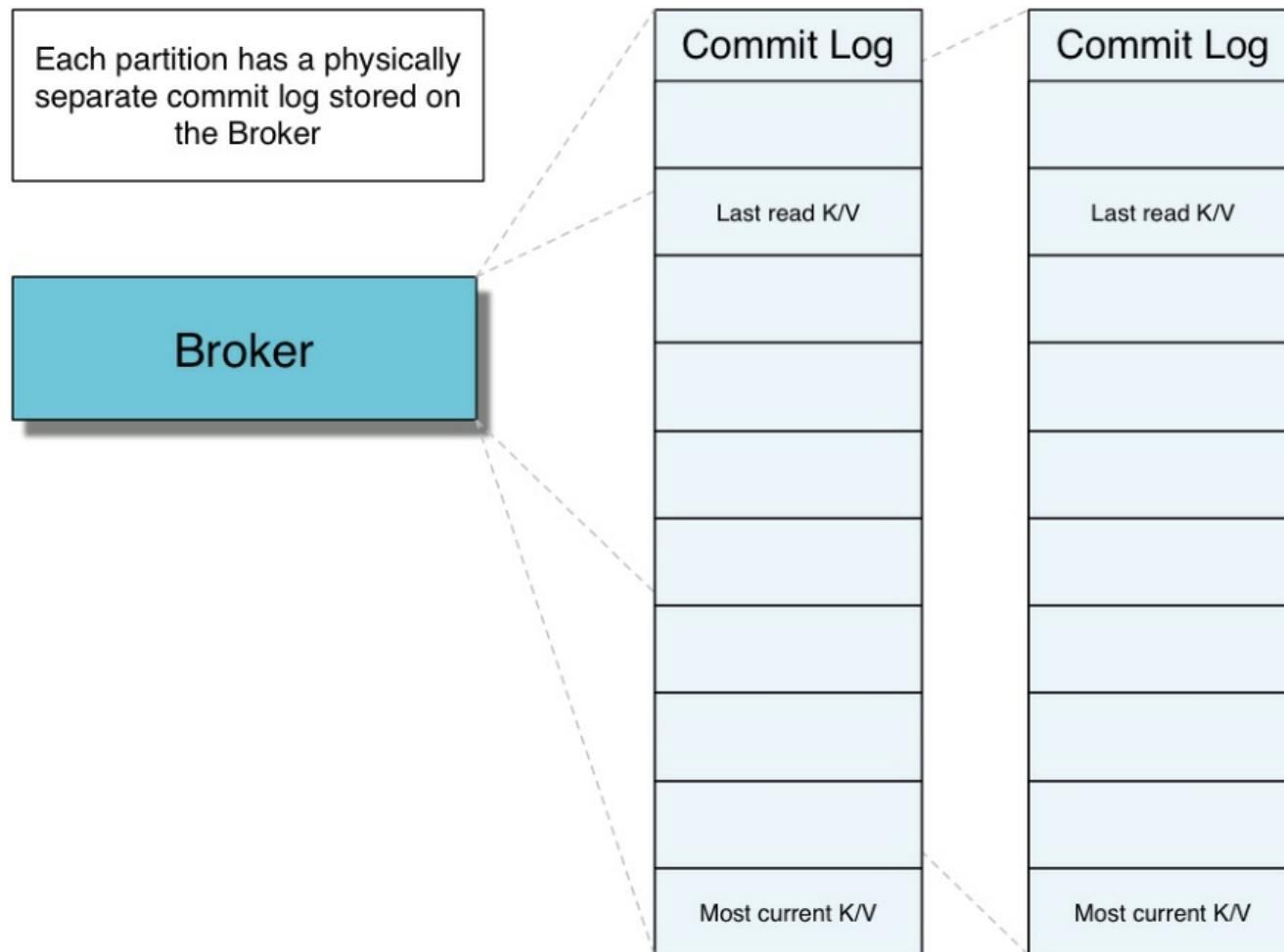
# The Commit Log for High Concurrency

---



# Partitions Are Stored as Separate Logs

---



# Kafka's Architecture

---

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

# Problems With our Current Model

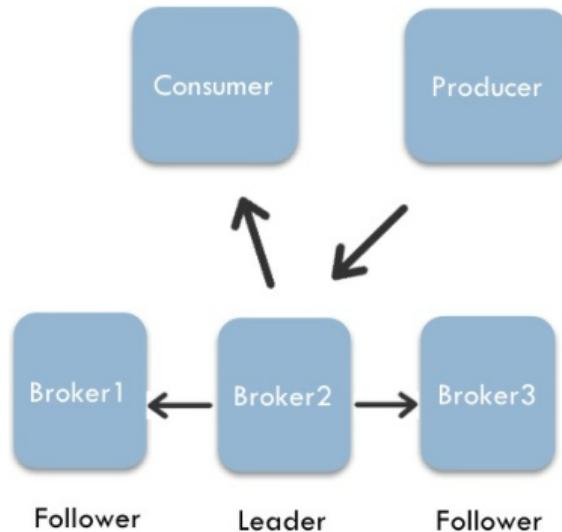
---

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

# Replication of Partitions

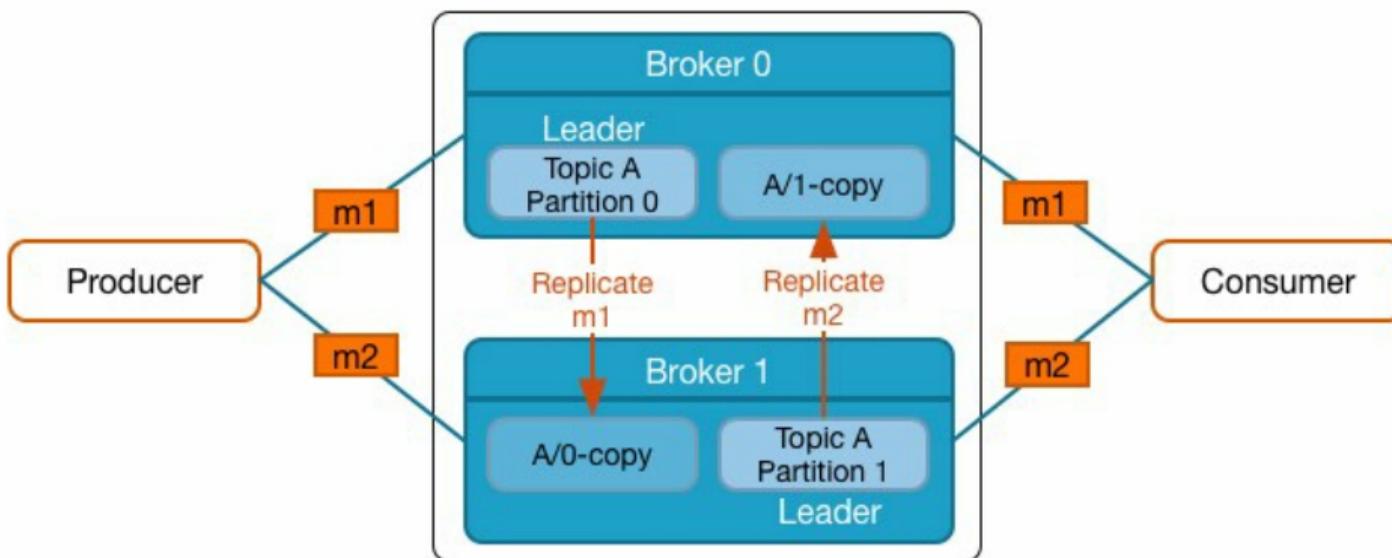
---

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



# Important: Clients Do Not Access Followers

- It is important to understand that Producers only write to the leader
- Likewise, Consumers *only* read from the leader
  - They do not read from the replicas
  - Replicas only exist to provide reliability in case of Broker failure
- In the diagram below, m1 hashes to Partition 0 and m2 hashes to Partition 1



- If a leader fails, the Kafka cluster will elect a new leader from among the followers

# In-Sync Replicas

---

- You may see information about “In-Sync Replicas” (ISRs) from some Kafka command-line tools
- ISRs are replicas which are up-to-date with the leader
  - If the leader fails, it is the list of ISRs which is used to elect a new leader
- Although this is more of an administration Topic, it helps to be familiar with the term ISR

# Managing Broker Failures

---

- **One Broker in the entire cluster is designated as the Controller**
  - Detects Broker failure/restart via ZooKeeper
- **Controller action on Broker failure**
  - Selects a new leader and updates the ISR list
  - Persists the new leader and ISR list to ZooKeeper
  - Sends the new leader and ISR list changes to all Brokers
- **Controller action on Broker restart**
  - Sends leader and ISR list information to the restarted Broker
- **If the Controller fails, one of the other Brokers will become the new Controller**

# Kafka's Architecture

---

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

# 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

# Preserve Message Ordering

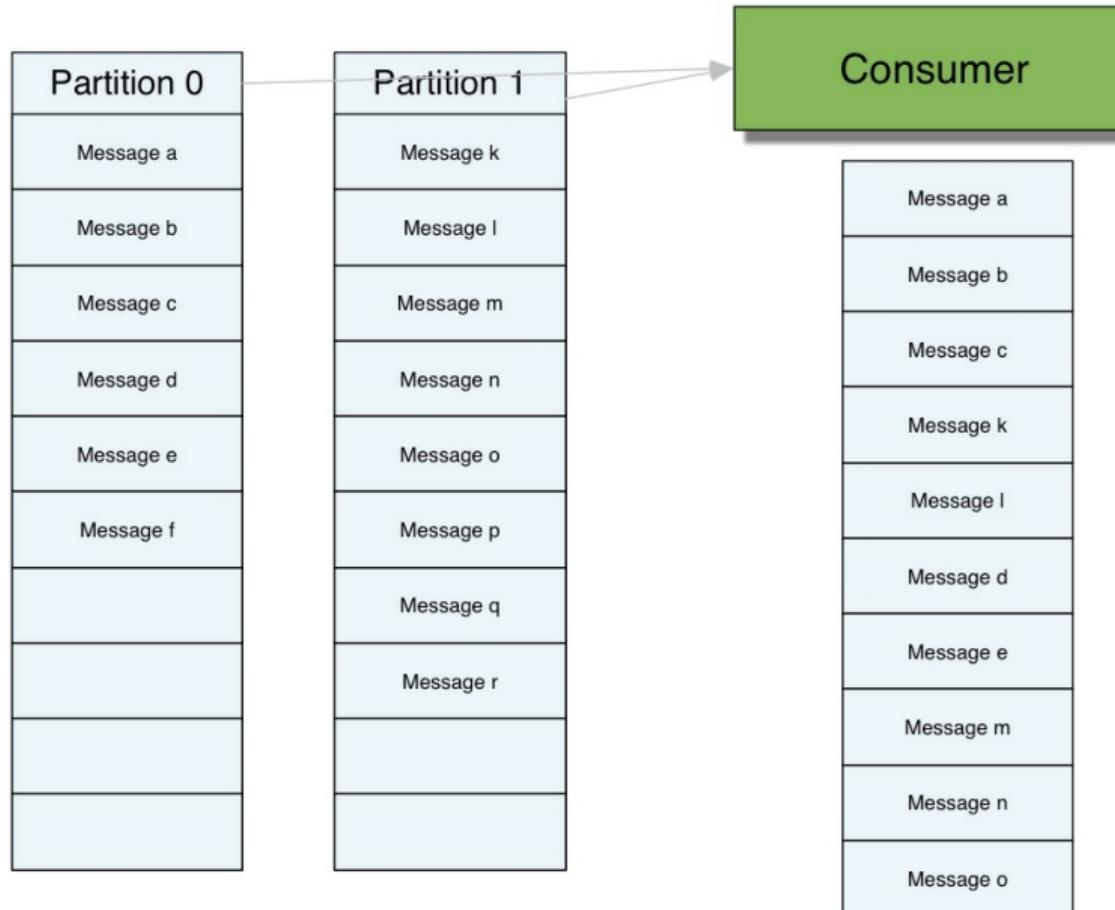
---

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

# An Important Note About Ordering

---

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

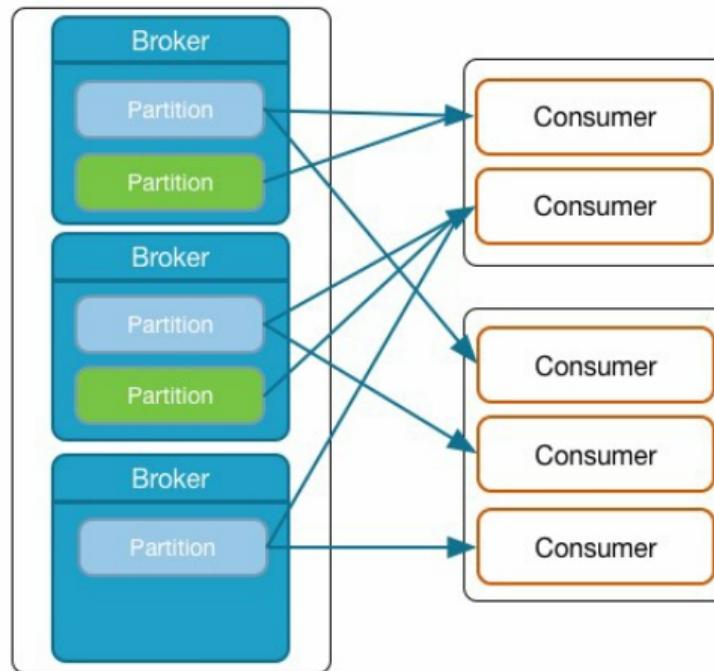


- Question: how can you preserve message order if the application requires it?

# Group Consumption Balances Load

---

- **Multiple Consumers in a *Consumer Group***
  - The group.id property is identical across all Consumers in the group
- **Consumers in the group are typically on separate machines**
- **Automatic failure detection and load rebalancing**



# Partition Assignment within a Consumer Group

---

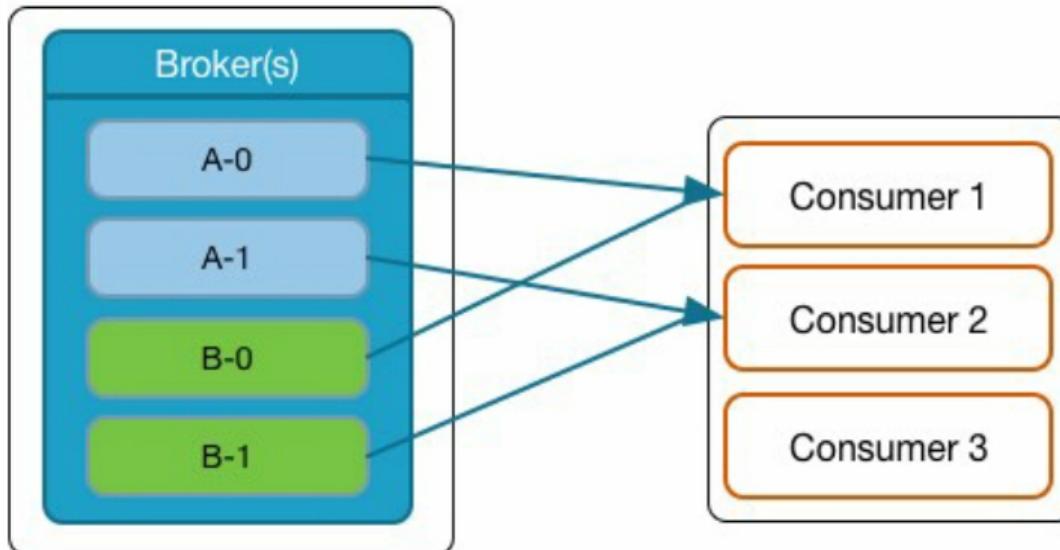
- **Partitions are ‘assigned’ to Consumers**
  - A single Partition is consumed by only one Consumer in any given Consumer Group
  - *i.e.*, you are guaranteed that messages with the same key will go to the same Consumer
  - Unless you change the number of partitions (see later)
  - `partition.assignment.strategy` in the Consumer configuration

# Partition Assignment Strategy: Range

---

- **Range (default)**

- Topic/Partition: topics A and B, each with two Partitions 0 and 1
- Consumer Group: Consumers c1, c2, c3
- c1: {A-0, B-0} c2: {A-1, B-1} c3: {}

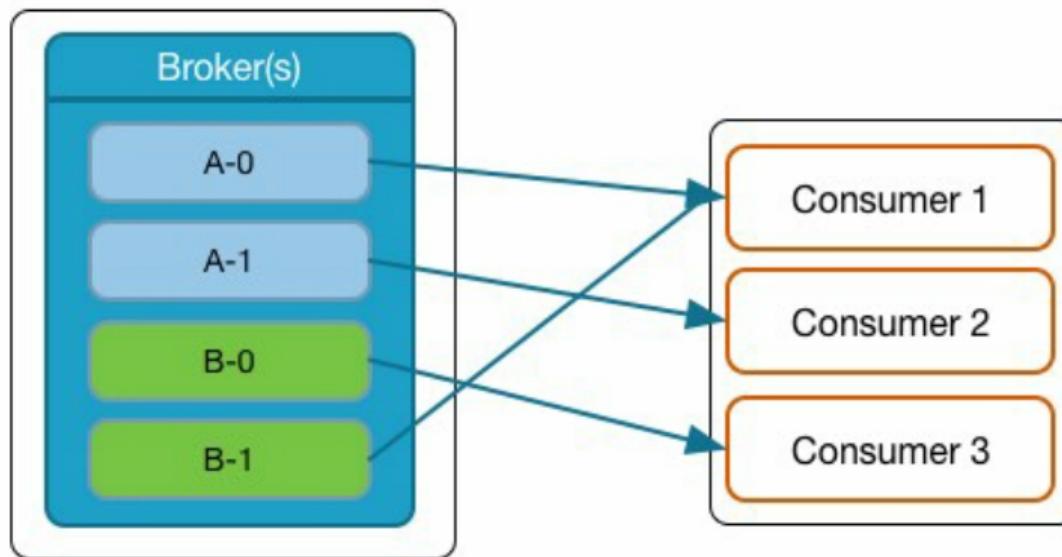


# Partition Assignment Strategy: RoundRobin

---

- **RoundRobin**

- c1: {A-0, B-1} c2: {A-1} c3: {B-0}



- **Partition assignment is automatically recomputed on changes in Partitions/Consumers**

# Partition Assignment Strategy: Sticky

---

- **Sticky assignment strategy was introduced in Kafka 0.11 (Confluent 3.3)**
- **Guarantees an assignment that is as balanced as possible**
- **Preserves existing Partition assignments to Consumers during reassignment to reduce overhead**
  - Kafka Consumers retain pre-fetched messages for Partitions assigned to them before a reassignment
  - Reduces the need to cleanup local Partition state between rebalances

# 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*
- *Replicas for Reliability*
- *Partitions and Consumer Groups for Scalability*
- **Hands-On Exercise: Consuming from Multiple Partitions**
- *Chapter Review*

# 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*
- *Replicas for Reliability*
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- **Chapter Review**

# Chapter Review

---

- **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**

# Developing With Kafka

Chapter 5



# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

>>> 05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# Developing With Kafka

---

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

- 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

---

- **Programmatically Accessing Kafka**
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

# The Kafka API

---

- **Since Kafka 0.9, Kafka includes 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 provides and supports client libraries for C/C++, Python, Go, and .NET**
- **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)

# Our Class Environment

---

- **During the course this week, we anticipate that you will be writing code either in Java...**
  - In which case, you will use Kafka's Java API
- **...or Python**
  - In which case, you will use the REST Proxy
- **If you wish to use some other programming language to access the REST proxy, you can do so**
  - Be aware that your instructor may not be familiar with your language of choice, though

# Developing With Kafka

---

- *Programmatically Accessing Kafka*
- **Writing a Producer in Java**
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

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

# Important Properties Elements (2)

---

Name	Description
<b>acks</b>	Number of acknowledgment the Producer requires the leader to have before considering the request complete. This controls the durability of records. <b>acks=0</b> : Producer will not wait for any acknowledgment from the server; <b>acks=1</b> : Producer will wait until the leader has written the record to its local log; <b>acks=all</b> : 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`

# Helper Classes

---

- Kafka includes helper classes **ProducerConfig**, **ConsumerConfig**, and **StreamsConfig**
  - Provide predefined constants for commonly configured properties
- Examples

```
// With helper class props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
// Without helper class
props.put("bootstrap.servers", "broker1:9092");
```

```
// With helper class
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
// Without helper class
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

```
// With helper class
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
// Without helper class
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

# Sending Messages to Kafka

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

① ProducerRecord can take an optional timestamp if you don't want to use current system time

② Alternatively:

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

# **send()** Does Not Block

---

- **The send() call is asynchronous**
  - It does not block; 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()**

# When Do Producers Actually Send?

---

- **A Producer send() returns immediately after it has added the message to a local buffer of pending record sends**
  - This allows the Producer to send records in batches for better performance
- **Then the Producer flushes multiple messages to the Brokers based on batching configuration parameters**
  - You can also manually flush by calling the Producer method flush()
- **Do not confuse the Producer method flush() in a Producer context with the term flush in a Broker context**
  - flush in a Broker context refers to when messages get written from the page cache to disk

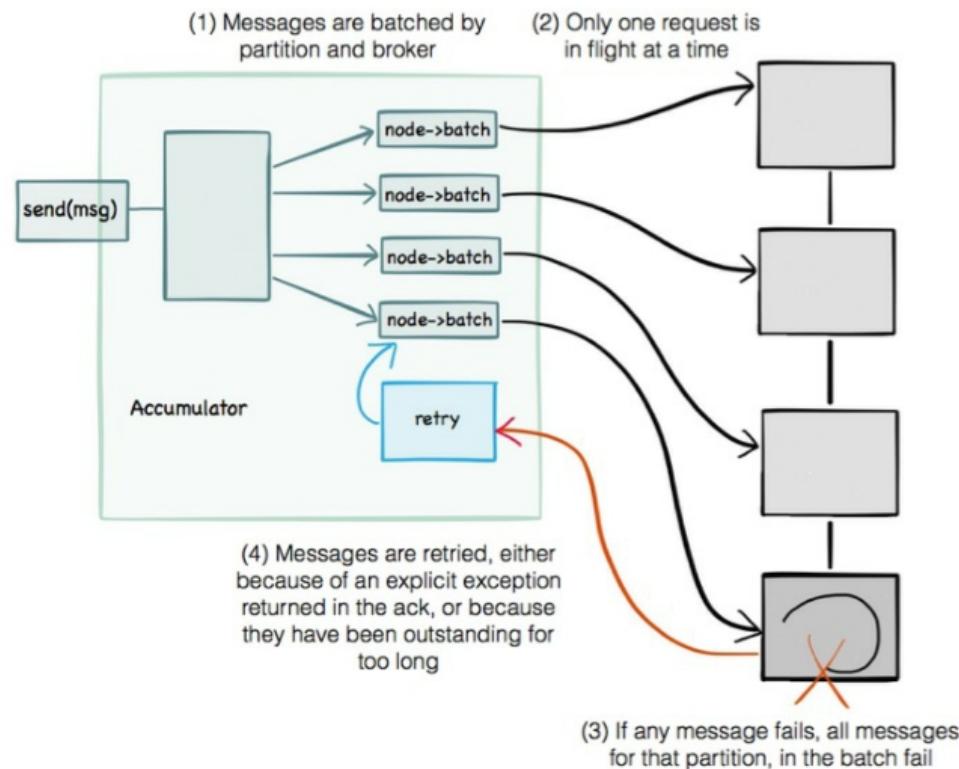
# Producer Retries

---

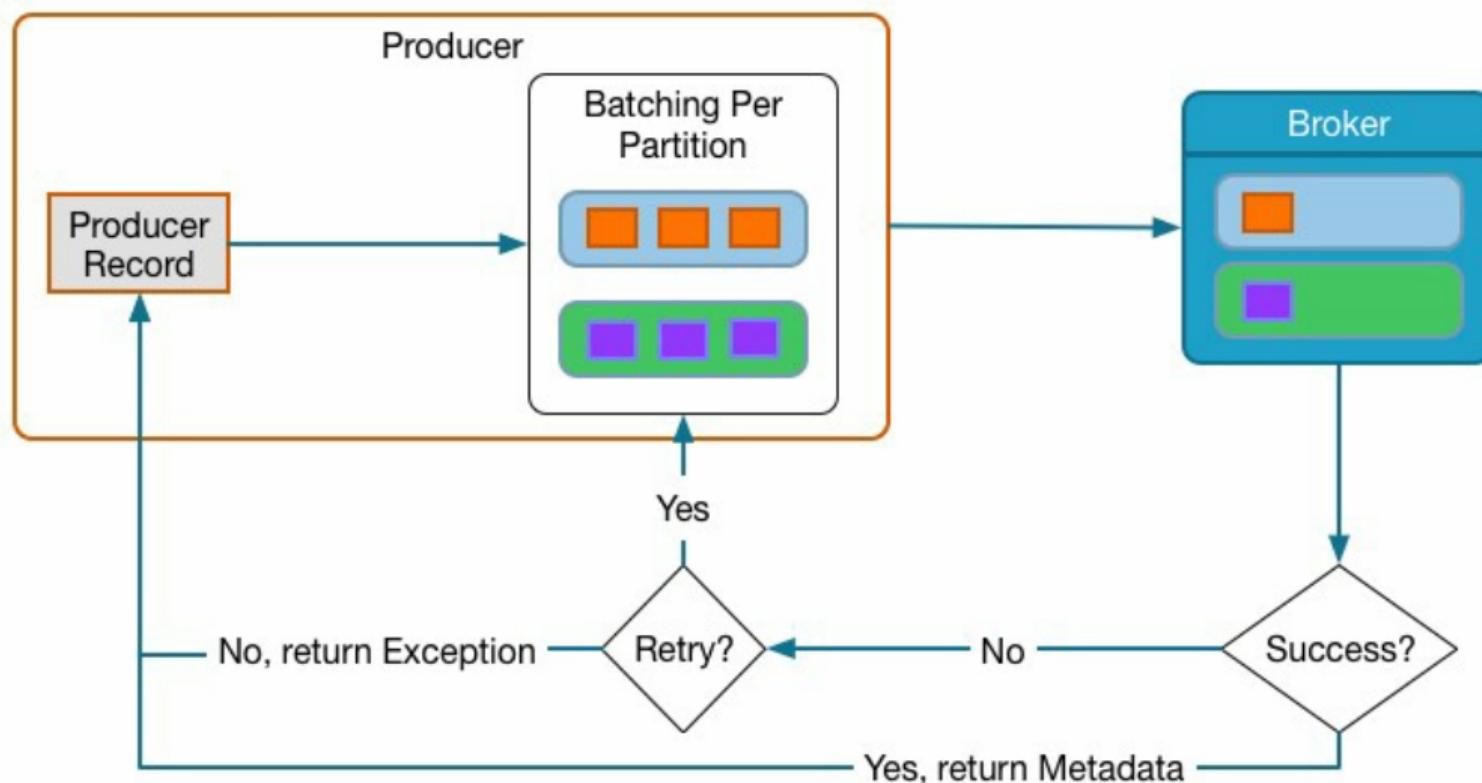
- **Developers can configure the retries configuration setting in the Producer code**
  - retries: how many times the Producer will attempt to retry sending records
    - Only relevant if acks is not 0
  - Hides transient failure
  - Ensure lost messages are retried rather than just throwing an error
    - retries: number times to retry (Default: 0)
    - retry.backoff.ms: pause added between retries (Default: 100)
    - For example, retry.backoff.ms=100 and retries=600 will retry for 60 seconds

# Preserve Message Send Order

- If retries > 0, message ordering could change
- To preserve message order, set `max.in.flight.requests.per.connection=1` (Default: 5)
  - May impact throughput performance because of lack of request pipelining



# Batching and Retries



# Performance Tuning Batching

---

- Producers can adjust batching configuration parameters
  - batch.size message batch size in bytes (Default: 16KB)
  - linger.ms time to wait for messages to batch together (Default: 0, *i.e.*, send immediately)
    - High throughput: large batch.size and linger.ms, or flush manually
    - Low latency: small batch.size and linger.ms
  - buffer.memory (Default: 32MB)
    - The Producer's buffer for messages to be sent to the cluster
    - Increase if Producers are sending faster than Brokers are acknowledging, to prevent blocking

# 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 metadata, java.lang.Exception exception)
```

- **Callback parameters**
  - metadata will be null if an error occurred
  - exception will be null if no error occurred

## send() and Callbacks (2)

---

- Parameters correlated to a particular record can be passed into the Callback's constructor
- Example code, with lambda function and closure instead of requiring a separate class definition

```
1 producer.send(record, (recordMetadata, e) -> {
2     if (e != null) {
3         e.printStackTrace();
4     } else {
5         System.out.println("Message String = " + record.value() + ", Offset = " + recordMetadata
6             .offset());
7     }
8});
```

# Closing the Producer

---

- **close(): blocks until all previously sent requests complete**

```
1 producer.close();
```

- **close(timeout, timeUnit): waits up to timeout for the producer to complete the sending of all incomplete requests**
  - If the producer is unable to complete all requests before the timeout expires, this method will fail any unsent and unacknowledged records immediately

# Developing With Kafka

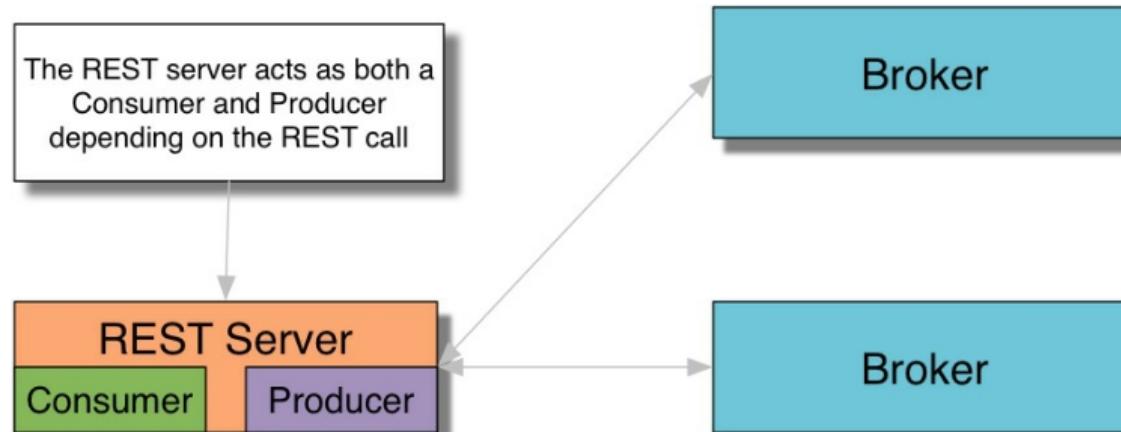
---

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- **Using the REST API to Write a Producer**
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

# About the REST Proxy

---

- The REST Proxy allows you to use HTTP to perform actions on the Kafka cluster
- The REST calls are translated into native Kafka calls
- This allows virtually any language to access Kafka
- Uses **POST** to send data to Kafka
  - Embedded formats: JSON, base64-encoded JSON, or Avro-encoded JSON
- Uses **GET** to retrieve data from Kafka



# A Python Producer Using the REST Proxy

---

```
1 #!/usr/bin/python
2
3 import requests
4 import base64
5 import json
6
7 url = "http://restproxy:8082/topics/my_topic"
8 headers = {
9     "Content-Type" : "application/vnd.kafka.binary.v1+json"
10    }
11 # Create one or more messages
12 payload = {"records":
13     [{"key":base64.b64encode("firstkey"),
14      "value":base64.b64encode("firstvalue")}
15     ]}}
16 # Send the message
17 r = requests.post(url, data=json.dumps(payload), headers=headers)
18 if r.status_code != 200:
19     print "Status Code: " + str(r.status_code)
20     print r.text
```

# Developing With Kafka

---

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- **Hands-On Exercise: Writing a Producer**
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

# Hands-On Exercise: Writing a Producer

---

- In this Hands-On Exercise, you will write a Kafka Producer either in Java or Python
- Please refer to the Hands-On Exercise Manual

# Developing With Kafka

---

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- **Writing a Consumer in Java**
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

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

# Creating the Properties and KafkaConsumer Objects

```
1 Properties props = new Properties();
2 props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
3 props.put(ConsumerConfig.GROUP_ID_CONFIG, "samplegroup");
4 props.put(ConsumerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringDeserializer.class);
5 props.put(ConsumerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringDeserializer.class);
6
7 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
8 consumer.subscribe(Arrays.asList("my_topic", "my_other_topic")); ① ②
```

- ① The Consumer can subscribe to as many Topics as it wishes. 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
- ② You may also use regular expressions (*i.e.*, Pattern) for topic subscription

# 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.

# Controlling the Number of Messages Returned

---

- By default, `poll()` fetches available messages from multiple partitions across multiple Brokers
  - 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
- The optional timeout parameter in the call to `poll()` is across all fetch requests
  - This controls the maximum amount of time in milliseconds that the Consumer will block if no new records are available
  - If records are available, it will return immediately
- In Kafka 0.10.0, `max.poll.records` was introduced
  - Property limits the total number of records retrieved in a single call to `poll()`

# Performance Tuning Consumer Fetch Requests

---

- **Consumption goal: high throughput or low latency?**
  - High throughput: send more data at a given time
  - Low latency: send immediately when data is available
- **fetch.min.bytes, fetch.max.wait.ms**
  - fetch.min.bytes (Default: 1)
    - Broker waits for messages to accumulate to this size batch before responding
  - fetch.max.wait.ms (Default: 500)
    - Broker will not wait longer than this duration before returning a batch
- **High throughput**
  - Large fetch.min.bytes, reasonable fetch.wait.max.ms
- **Low latency**
  - fetch.min.bytes=1

# 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 maximum size for a batch of messages that the Broker can receive from a Producer**
  - Broker: `message.max.bytes` (Default: 1MB)
  - Topic override: `max.message.bytes` (Default: 1MB)

# 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
- It is important to note that `KafkaConsumer` is not thread-safe

```
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     }  
7 } finally {  
8     consumer.close();  
9 }
```

# Developing With Kafka

---

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- **Using the REST API to Write a Consumer**
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

# A Python Consumer Using the REST API (1)

---

```
1 #!/usr/bin/python
2
3 import requests
4 import base64
5 import json
6 import sys
7
8 # Base URL for interacting with REST server
9 baseUrl = "http://restproxy:8082/consumers/group1" ①
10
11 # Create the Consumer instance
12 print "Creating consumer instance"
13 payload = {
14     "format": "binary"
15 }
16 headers = {
17     "Content-Type" : "application/vnd.kafka.v1+json"
18 }
```

① We are creating a Consumer instance in a Consumer Group called group1

# A Python Consumer Using the REST API (2)

---

```
19 r = requests.post(baseurl, data=json.dumps(payload), headers=headers)
20
21 if r.status_code != 200:
22     print "Status Code: " + str(r.status_code)
23     print r.text
24     sys.exit("Error thrown while creating consumer")
25
26 # Base URI is used to identify the consumer instance
27 base_uri = r.json()["base_uri"]
```

# A Python Consumer Using the REST API (3)

---

```
28 # Get the message(s) from the Consumer
29 headers = {
30     "Accept" : "application/vnd.kafka.binary.v1+json"
31 }
32
33 # Request messages for the instance on the Topic
34 r = requests.get(base_uri + "/topics/my_topic", headers=headers, timeout=20)
35
36 if r.status_code != 200:
37     print "Status Code: " + str(r.status_code)
38     print r.text
39     sys.exit("Error thrown while getting message")
```

# A Python Consumer Using the REST API (4)

---

```
40 # Output all messages
41 for message in r.json():
42     if message["key"] is not None:
43         print "Message Key:" + base64.b64decode(message["key"])
44         print "Message Value:" + base64.b64decode(message["value"])
45
46 # When we're done, delete the Consumer
47 headers = {
48     "Accept" : "application/vnd.kafka.v1+json"
49 }
50
51 r = requests.delete(base_uri, headers=headers)
52
53 if r.status_code != 204:
54     print "Status Code: " + str(r.status_code)
55     print r.text
```

# Developing With Kafka

---

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- **Hands-On Exercise: Writing a Basic Consumer**
- *Chapter Review*

# 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

---

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- **Chapter Review**

# Chapter Review

---

- **The Kafka API provides Java clients for Producers and Consumers**
- **Client libraries for other languages are available**
  - Confluent provides and supports client libraries for C/C++, Python, Go, and .NET
- **Confluent's REST Proxy allows other languages to access Kafka without the need for native client libraries**

# **More Advanced Kafka Development**

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

**>>> 06: More Advanced Kafka Development**

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# More Advanced Kafka Development

---

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

- How to enable exactly once semantics (EOS)
- 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

---

- **Exactly Once Semantics**
- *Specifying Offsets*
- *Consumer ‘Liveness’ and Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Review*

# Motivation for Exactly Once Semantics

---

- Write real-time, mission-critical streaming applications that require guarantees that data is processed “exactly once”
  - Complements the “at most once” or “at least once” semantics that exist today
- Exactly Once Semantics (EOS) bring strong transactional guarantees to Kafka
  - Prevents duplicate messages from being processed by client applications
  - Even in the event of client retries and Broker failures
- Use cases: tracking ad views, processing financial transactions, etc.

# EOS in Kafka

---

- EOS was introduced in Kafka 0.11 (Confluent 3.3)
- Supported with the new Java Producer and Consumer
- Supported with the Kafka Streams API
- Transaction Coordinator manages transactions and assign Producer IDs
- Updated log format with sequence numbers and Producer IDs

# Enabling EOS on the Producer Side (1)

---

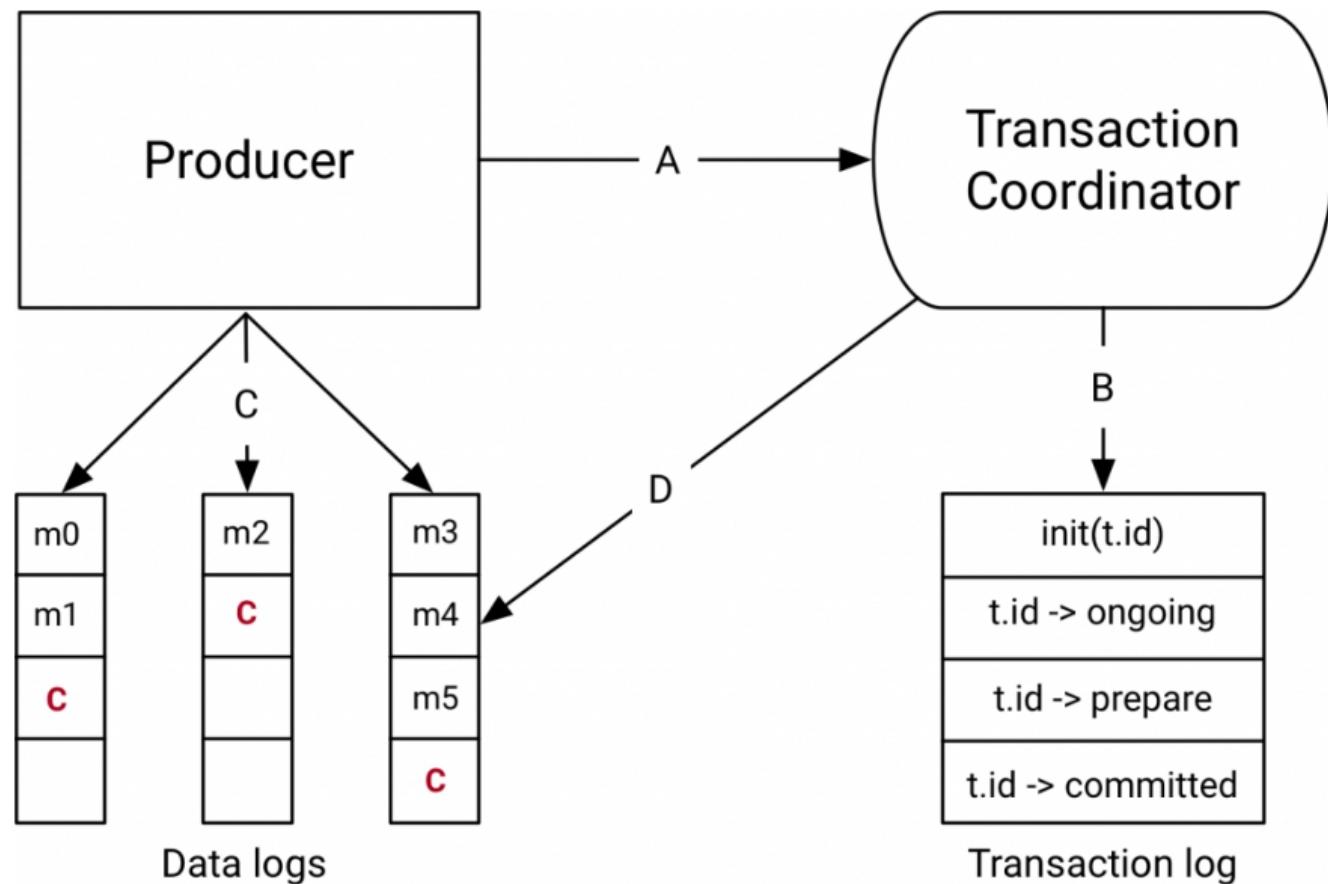
- **Configure the Producer send operation to be idempotent**
  - An idempotent operation is one which can be performed many times without causing a different effect than only being performed once
  - Removes the possibility of duplicate messages being delivered to a particular Topic Partition due to Producer or Broker errors
  - Set enable.idempotence to true (Default: false)
- **A Producer writes atomically across multiple Partitions using "transactional" messages**
  - Either all or no messages in a batch are visible to Consumers
  - Use the new transactions API
- **Allow reliability semantics to span multiple producer sessions**
  - Set transactional.id to guarantee that transactions using the same transactional ID have completed prior to starting new transactions

# Enabling EOS on the Producer Side (2)

---

```
1 producer.initTransactions();
2 try {
3     producer.beginTransaction();
4     producer.send(record1);
5     producer.send(record2);
6     producer.commitTransaction();
7 } catch(ProducerFencedException e) {
8     producer.close();
9 } catch(KafkaException e) {
10    producer.abortTransaction();
11 }
```

# Producer Transactions across Partitions



# Enabling EOS on the Consumer Side

---

- **Configure the Consumer to read committed transactional messages**
  - Set isolation.level to read\_committed (Default: read\_uncommitted)
    - read\_committed: reads only committed transactional messages and all non-transactional messages
    - read\_uncommitted: reads all transactional messages and all non-transactional messages

# More Advanced Kafka Development

---

- *Exactly Once Semantics*
- **Specifying Offsets**
- *Consumer ‘Liveness’ and Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Review*

# Determining the Offset When a Consumer Starts

---

- **The Consumer property `auto.offset.reset` determines what to do if there is no valid offset in Kafka for the Consumer's Consumer Group**
  - When a particular Consumer Group starts the first time
  - If the Consumer offset is less than the smallest offset
  - If the Consumer offset is greater than the last offset
- **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 ways to view offsets and dynamically change the offset from which the Consumer will read
- View offsets
  - `position(TopicPartition)` provides the offset of the next record that will be fetched
  - `offsetsForTimes(Map<TopicPartition,Long> timestampsToSearch)` looks up the offsets for the given Partitions by timestamp
- Change offsets
  - `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 consumer.subscribe(Arrays.asList("my_topic"));
2 consumer.poll(0);
3 consumer.seekToBeginning(consumer.assignment()); ①
```

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

# Changing the Offset Within the Consumer (3)

- For example, to reset the offset to a particular timestamp, you could do something like:

```
1 for (TopicPartition topicPartition : partitionSet) {  
2     timestampsToSearch.put(topicPartition, MY_TIMESTAMP); ①  
3 }  
4  
5 Map<TopicPartition, OffsetAndTimestamp> result = consumer.offsetsForTimes(timestampsToSearch);  
②  
6  
7 for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry : result.entrySet()) {  
8     consumer.seek(entry.getKey(), entry.getValue().offset()); ③  
9 }
```

- ① Add each Partition and timestamp to the HashMap
- ② Get the offsets for each Partition
- ③ Seek to the specified offset for each Partition

Question: what is a use case where this would be useful?

# More Advanced Kafka Development

---

- *Exactly Once Semantics*
- *Specifying Offsets*
- **Consumer ‘Liveness’ and Rebalancing**
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Review*

# Consumer ‘Liveness’

---

- **The Consumer communicates to the Brokers regularly to let them know that it is still alive**
- **If a Consumer is believed to be dead, it is removed from the Consumer Group and a Consumer rebalance is performed**
  - Reallocates all Partitions of a Topic to live members of the Consumer Group

# Consumer Heartbeats for ‘Liveness’

---

- As of Kafka 0.10.1.0, heartbeats are sent in a background thread, separate from calls to poll
- New defaults
  - session.timeout.ms now defaults to 10 seconds
  - max.poll.records now defaults to 500
- To avoid issues where a Consumer stops processing records but the background thread is still running, poll must still be called periodically
  - max.poll.interval.ms: maximum allowable interval between calls to poll  
(Default: 5 minutes)

# Pausing Record Retrieval

---

- **There are situations where Consumers may wish to temporarily stop consumption of new messages**
  - The Consumer can pause and resume consumption on a per-Partition basis
  - To maintain liveness, poll must be called periodically, even in Kafka 0.10.1.0 and above
- **Use cases**
  - Prior to 0.10.1.0, continue heartbeating during long message processing times
  - Processing and joining data from two topics, where one lags behind another
- **How pause works**
  - Each pause call takes a Collection of TopicPartition objects
  - While paused, calls to poll return no new messages for those Partitions
- **Caution: In this scenario, you will probably need to manually manage topic offsets**

# Partition Assignments for Consumers

---

- We have said that all the data from a particular Partition will go to the same Consumer
- This is true *as long as* the number of Consumers in a Consumer Group does not change
- If the number of Consumers changes, a Partition rebalance occurs
  - Partition ownership is moved around between the Consumers to spread the load evenly
  - No guarantees that a Consumer will get the same or different Partition
- Consumers cannot consume messages during the rebalance, so this results in a short pause in message consumption

# Consumer Rebalancing

---

- **Consumer rebalances are initiated when**
  - A Consumer leaves the Consumer group (either by failing to send a timely heartbeat or by explicitly requesting to leave)
  - A new Consumer joins the Consumer Group
  - A Consumer changes its Topic subscription
  - The Consumer Group notices a change to the Topic metadata for any subscribed Topic (*e.g.* an increase in the number of Partitions)
- **Rebalancing does not occur if**
  - A Consumer calls pause
- **During rebalance, consumption is paused**
- **Question: could adding a Consumer to a Consumer Group cause Partition assignment to change?**

# The Case For and Against Rebalancing

---

- **Typically, Partition rebalancing is a good thing**
  - Allows you to add more Consumers to a Consumer Group dynamically, without having to restart all the other Consumers in the group
  - Automatically handles situations where a Consumer in the Consumer Group fails
- **However a rebalance is like a fresh restart**
  - Consumers may or may not get a different set of Partitions
    - If your Consumer is relying on getting all data from a particular Partition, this could be a problem
  - A previously-paused Partition is no longer paused
- **Managing rebalances where Partition assignment matters**
  - Option 1: only have a single Consumer for the entire topic
  - Option 2: provide a ConsumerRebalanceListener when calling subscribe()
    - Implement onPartitionsRevoked and onPartitionsAssigned methods

# More Advanced Kafka Development

---

- *Exactly Once Semantics*
- *Specifying Offsets*
- *Consumer ‘Liveness’ and Rebalancing*
- **Manually Committing Offsets**
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Review*

# 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

# Manually Committing Offsets From the REST Proxy

---

- It is possible to manually commit offsets from the REST Proxy

```
1 payload = {  
2     "format": "binary",  
3     # Manually/Programmatically commit offset  
4     "auto.commit.enable": "false"  
5 }  
6  
7 headers = {  
8 "Content-Type" : "application/vnd.kafka.v1+json"  
9 }  
10  
11 r = requests.post(baseurl, data=json.dumps(payload), headers=headers)  
12  
13 # Commit the offsets  
14 if shouldCommit() == True:  
15     r = requests.post(base_uri + "/offsets", headers=headers, timeout=20)  
16     if r.status_code != 200:  
17         print "Status Code: " + str(r.status_code)  
18         print r.text  
19         sys.exit("Error thrown while committing")  
20     print "Committed"
```

## 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**

## Quiz: Question

---

- A team responsible for consuming data from a Kafka cluster is reporting that the cluster is not properly managing offsets. They describe that when they add Consumers to a Consumer Group, all the Consumers see some previously-consumed messages. What is happening?

# Quiz: Answer

---

- **When the Consumer Group membership changes, it triggers a rebalance**
  - Existing Partitions may be reassigned to different Consumers
- **Offsets are managed per Partition not per Consumer**
  - This works only if the Consumer code is written well and the offsets are committed upon consumption
- **Action:**
  - Review Consumer client code for committing offsets during consumption
    - Commit intervals
    - commitSync() vs commitAsync()
  - Consumers should implement ConsumerRebalanceListener code
    - Consumer can commit offsets in reaction to receiving an onPartitionsRevoked event

# More Advanced Kafka Development

---

- *Exactly Once Semantics*
- *Specifying Offsets*
- *Consumer ‘Liveness’ and Rebalancing*
- *Manually Committing Offsets*
- **Partitioning Data**
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Review*

# 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
- 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"))  
14            ① 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>("my\_topic", 0, key, value);
    - Will write the message to Partition 0

- Discussion: Which method is preferable?

# More Advanced Kafka Development

---

- *Exactly Once Semantics*
- *Specifying Offsets*
- *Consumer ‘Liveness’ and Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- **Message Durability**
- *Hands-On Exercise: Accessing Previous Data*
- *Chapter Review*

# 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

# Delivery Acknowledgment

---

- The **acks** configuration parameter determines the behavior of the Producer when sending messages
- Use this to configure the durability of messages being sent

<b>acks=0</b>	Producer will not wait for any acknowledgment. The message is placed in the Producer's buffer, and immediately considered sent
<b>acks=1</b>	The Producer will wait until the leader acknowledges receipt of the message
<b>acks=all</b>	The leader will wait for acknowledgment from all in-sync replicas before reporting the message as delivered to the Producer

- **min.insync.replicas with acks=all**
  - defines the minimum number of replicas in ISR needed to satisfy a produce request

# Important Timeouts

---

- **request.timeout.ms**
  - Maximum time waiting for the response to a request
  - After timeout, the client will resend or throw an error if retries are exhausted
    - Recall that retries defaults to 0 and is only relevant if acks is not 0
  - Default: 30s
- **max.block.ms**
  - The send() call waits this time until the message can be successfully put into the bufferpool
  - After timeout, it will throw a TimeoutException
  - Default: 60s

# More Advanced Kafka Development

---

- *Exactly Once Semantics*
- *Specifying Offsets*
- *Consumer ‘Liveness’ and Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- **Hands-On Exercise: Accessing Previous Data**
- *Chapter Review*

# 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

---

- *Exactly Once Semantics*
- *Specifying Offsets*
- *Consumer ‘Liveness’ and Rebalancing*
- *Manually Committing Offsets*
- *Partitioning Data*
- *Message Durability*
- *Hands-On Exercise: Accessing Previous Data*
- **Chapter Review**

# Chapter Review

---

- 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
- You can configure the reliability of message delivery by specifying different values for the acks configuration parameter

# **Schema Management In Kafka**

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

**>>> 07: Schema Management In Kafka**

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# 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 Review*

# 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 Review*

# 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
<b>record</b>	A user-defined field comprising one or more simple or complex data types, including nested records
<b>enum</b>	A specified set of values
<b>union</b>	Exactly one value from a specified set of types
<b>array</b>	Zero or more values, each of the same type
<b>map</b>	Set of key/value pairs; key is always a <b>string</b> , value is the specified type
<b>fixed</b>	A fixed number of bytes

- **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

---

- **Avro schemas may evolve as updates to code happen**
  - Schema registry allows schema evolution
- **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

# Compatibility Examples

---

- Consider a schema written with the following fields

```
{ "name": "suit", "type": "string"},  
{ "name": "card", "type": "string"}
```

- Backward compatibility: Consumer is expecting the following schema and assumes default for omitted size field

```
{ "name": "suit", "type": "string"},  
{ "name": "card", "type": "string"},  
{ "name": "size", "type": "string", "default": "" }
```

- Forward compatibility: Consumer is expecting the following schema and ignores additional card field

```
{ "name": "suit", "type": "string"}
```

- Question: What could be an example of Full compatibility using the fields above?

# Schema Management In Kafka

---

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

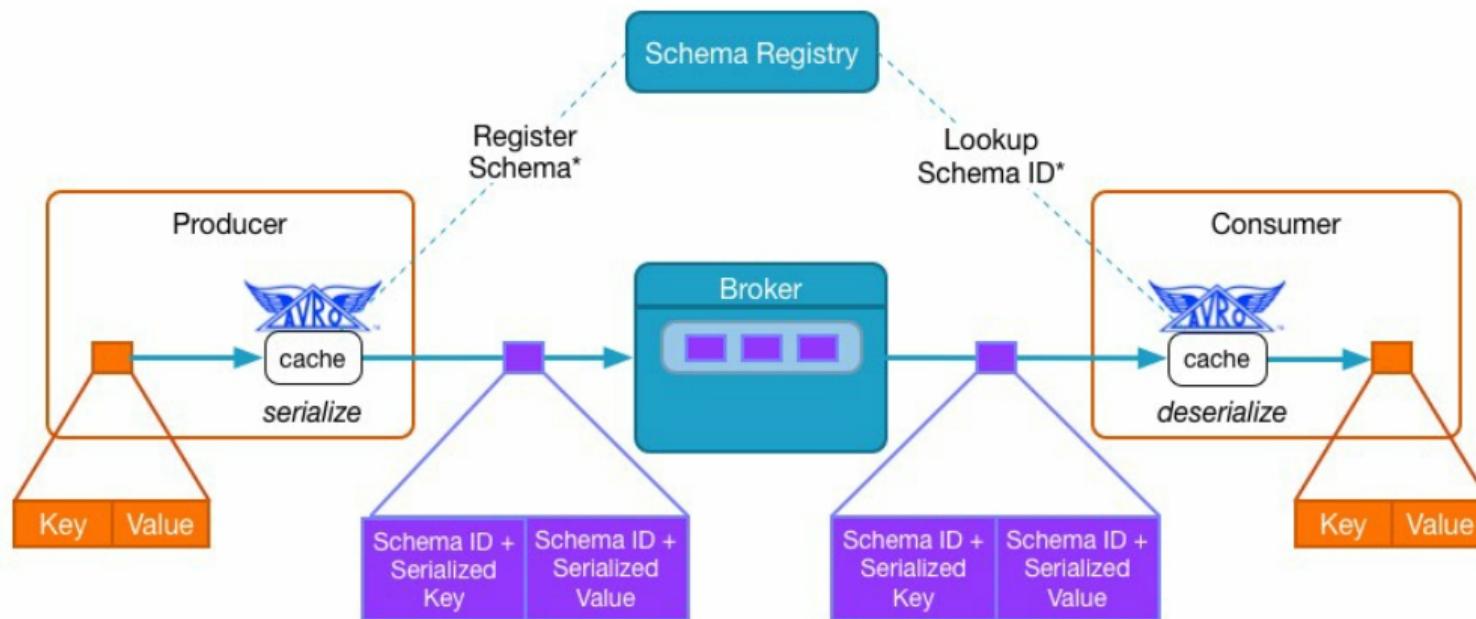
# What Is the Schema Registry?

---

- **Schema registry provides centralized management of schemas**
  - Stores a versioned history of all schemas
  - Provides a RESTful interface for storing and retrieving Avro schemas
  - Checks schemas and throws an exception if data does not conform to the schema
  - Allows evolution of schemas according to the configured compatibility setting
- **Sending the Avro schema with each message would be inefficient**
  - Instead, a globally unique ID representing the Avro schema is sent with each message
- **The Schema Registry stores schema information in a special Kafka topic**
- **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

# Schema Registration and Dataflow

- The message key and value can be independently serialized
- Producers serialize data and prepend the schema ID
- Consumers use the schema ID to deserialize the data
- Schema Registry communication is only on the first message of a new schema
  - Producers and Consumers cache the schema/ID mapping for future messages



# Client Support for the Schema Registry (1)

---

- **Java client**

- New schemas automatically registered

```
Properties newProperties = new Properties();
newProperties.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://schemaregistry1:8081");
```

- **Python client (confluent-kafka-python)**

- Support for Schema Registry introduced in Kafka 0.10.2 (Confluent 3.2)

```
value_schema = avro.load('ValueSchema.avsc')
key_schema = avro.load('KeySchema.avsc')
avroProducer = AvroProducer({'bootstrap.servers': 'broker101', 'schema.registry.url':
'http://schemaregistry1:9001'}, default_key_schema=key_schema, default_value_schema=value_schema)
```

# Client Support for the Schema Registry (2)

---

- Other clients

- Use the Schema Registry REST API to manually pre-register schemas and use the IDs in the requests

```
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"string\"}"}' \ http://schemaregistry1:8081/subjects/Kafka-
key/versions
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"string\"}"}' \ http://schemaregistry1:8081/subjects/Kafka-
value/versions
$ curl -X GET http://localhost:8081/schemas/ids/1
```

# Different Versions of Schemas in the Same Topic

---

- **Different versions of 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://schemaregistry1:8081/config/my_topic
```

# Java Avro Producer example

```
1 Properties props = new Properties();
2 props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "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(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
10            "http://schemaregistry1:8081");
11 // Create the producer expecting Avro objects
12 KafkaProducer<Object, Object> avroProducer = new KafkaProducer<Object, Object>(props);
13 // Create the Avro objects for the key and value
14 CardSuit suit = new CardSuit("spades");
15 SimpleCard card = new SimpleCard("spades", "ace");
16 // Create the ProducerRecord with the Avro objects and send them
17 ProducerRecord<Object, Object> record = new
18 ProducerRecord<Object, Object>(
19     "my_avro_topic", suit, card);
20 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(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
5         props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
6         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
7             "io.confluent.kafka.serializers.KafkaAvroDeserializer");
8         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
9             "io.confluent.kafka.serializers.KafkaAvroDeserializer");
10        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
11            "http://schemaregistry1:8081");
12        props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "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",
21                    record.offset(), record.key().getSuit(),
22                    record.value().getCard());
23            }
24        }
25    }
26 }
```

# Python Avro Producer Example

---

```
1 # Read in the Avro files
2 key_schema = open("my_key.avsc", 'rU').read()
3 value_schema = open("my_value.avsc", 'rU').read()
4
5 producerurl = "http://kafkarest1:8082/topics/my_avro_topic"
6 headers = {
7     "Content-Type" : "application/vnd.kafka.avro.v1+json"
8 }
9 payload = {
10    "key_schema": key_schema,
11    "value_schema": value_schema,
12    "records":
13    [
14        {
15            "key": {"suit": "spades"},
16            "value": {"suit": "spades", "card": "ace"}
17        }
18    ]
19 }
20
21 # Send the message
22 r = requests.post(producerurl, data=json.dumps(payload), headers=headers)
23 if r.status_code != 200:
24     print "Status Code: " + str(r.status_code)
25     print r.text
```

# Python Avro Consumer Example

---

```
1 # Get the message(s) from the consumer
2 headers = {
3     "Accept" : "application/vnd.kafka.avro.v1+json"
4 }
5 # Request messages for the instance on the topic
6 r = requests.get(base_uri + "/topics/my_avro_topic", headers=headers, timeout=20)
7 if r.status_code != 200:
8     print "Status Code: " + str(r.status_code)
9     print r.text
10    sys.exit("Error thrown while getting message")
11 # Output all messages
12 for message in r.json():
13     keysuit = message["key"]["suit"]
14     valuesuit = message["value"]["suit"]
15     valuecard = message["value"]["card"]
16     # Do something with the data
```

# Command-line Consumer Example

---

```
$ kafka-console-consumer --bootstrap-server broker1:9092 \  
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 Review*

# 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 Review**

# Chapter Review

---

- 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

# **Connect for Data Movement**

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

**>>> 08: Kafka Connect for Data Movement**

09: Basic Kafka Administration

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# Connect for Data Movement

---

- In this chapter you will learn:

- The motivation for Kafka Connect
- What standard Connectors are provided
- The differences between standalone and distributed mode
- How to configure and use Kafka Connect
- How Kafka Connect compares to writing your own data transfer system

# Kafka Connect for Data Movement

---

- **The Motivation for Kafka Connect**
- *Types of Connectors*
- *Kafka Connect Implementation*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Comparing Kafka Connect with Other Options*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

# What is Kafka Connect?

---

- **Kafka Connect is a framework for streaming data between Apache Kafka and other data systems**
- **Kafka Connect is open source, and is part of the Apache Kafka distribution**
- **It is simple, scalable, and reliable**



# Example Use Cases

---

- **Example use cases for Kafka Connect include:**
  - Stream an entire SQL database into Kafka
  - Stream Kafka Topics into HDFS for batch processing
  - Stream Kafka Topics into Elasticsearch for secondary indexing
  - ...

# Why Not Just Use Producers and Consumers?

---

- Internally, Kafka Connect is just a Kafka client using the standard Producer and Consumer APIs
- Kafka Connect has benefits over ‘do-it-yourself’ Producers and Consumers:
  - Off-the-shelf, tested Connectors for common data sources are available
  - Features fault tolerance and automatic load balancing when running in distributed mode
  - No coding required
    - Just write configuration files for Kafka Connect
  - Pluggable/extendable by developers

# Connect Basics

---

- ***Connectors*** are logical jobs responsible for managing the copying of data between Kafka and another system
- **Connector Sources** read data *from* an external data system into Kafka
  - Internally, a connector source is a Kafka Producer client
- **Connector Sinks** write Kafka data *to* an external data system
  - Internally, a connector sink is a Kafka Consumer client

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- **Types of Connectors**
- *Kafka Connect Implementation*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Comparing Kafka Connect with Other Options*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

# Off-The-Shelf Connectors

---

- **Confluent Open Source ships with commonly used Connectors**
  - JDBC
  - HDFS
  - Elasticsearch
  - AWS S3
  - FileStream
- **Confluent Enterprise includes an additional connector**
  - Replicator
- **Many other certified Connectors are available**
  - See <https://www.confluent.io/product/connectors/>

# JDBC Source Connector: Overview

---

- JDBC Source periodically polls a relational database for new or recently modified rows
  - Creates a record for each row, and Produces that record as a Kafka message
- 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 several ways:

Incremental query made	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

# JDBC Source Connector: Configuration

Parameter	Description
<b>connection.url</b>	The JDBC connection URL for the database
<b>topic.prefix</b>	The prefix to prepend to table names to generate the Kafka Topic name
<b>mode</b>	The mode for detecting table changes. Options are <b>bulk</b> , <b>incrementing</b> , <b>timestamp</b> , <b>timestamp+incrementing</b>
<b>query</b>	The custom query to run, if specified
<b>poll.interval.ms</b>	The frequency in milliseconds to poll for new data in each table
<b>table.blacklist</b>	A list of tables to ignore and not import. If specified, <b>tables.whitelist</b> cannot be specified
<b>table.whitelist</b>	A list of tables to import. If specified, <b>tables.blacklist</b> cannot be specified

- Note: This is not a complete list. See <http://docs.confluent.io>

# HDFS Sink Connector: Overview

---

- Continuously polls from Kafka and writes to HDFS (Hadoop Distributed File System)
- Integrates with Hive
  - Auto table creation
  - Schema evolution with Avro
- Works with secure HDFS and the Hive Metastore, using Kerberos
- Provides exactly once delivery
- Data format is extensible
  - Avro, Parquet, custom formats
- Pluggable Partitioner, supporting:
  - Kafka Partitioner (default)
  - Field Partitioner
  - Time Partitioner
  - Custom Partitioners

# FileStream Connector

---

- **FileStream Connector acts on a local file**
  - Local file Source Connector: tails local file and sends each line as a Kafka message
  - Local file Sink Connector: Appends Kafka messages to a local file

# New Connectors

---

- You can install a new Connector
  - Package the Connector in a JAR file
- Install the file on all of the Kafka Connect worker machines
- As of Kafka 0.11.0: install connectors as plugins
  - There is library isolation between plugins

```
plugin.path=/path/to/my/plugins
```

- Prior to Kafka 0.11.0: place the JAR in a path specified by CLASSPATH
  - There is no library isolation between connectors

```
$ export CLASSPATH="$CLASSPATH:/path/to/my/connectors/*"
```

# Kafka Connect for Data Movement

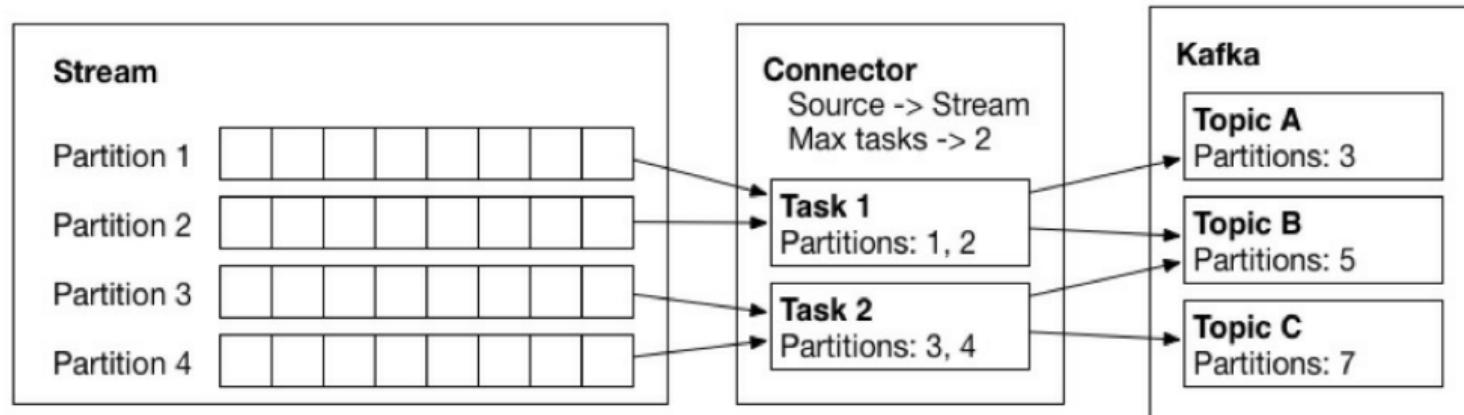
---

- *The Motivation for Kafka Connect*
- *Types of Connectors*
- **Kafka Connect Implementation**
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Comparing Kafka Connect with Other Options*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

# Providing Parallelism and Scalability

---

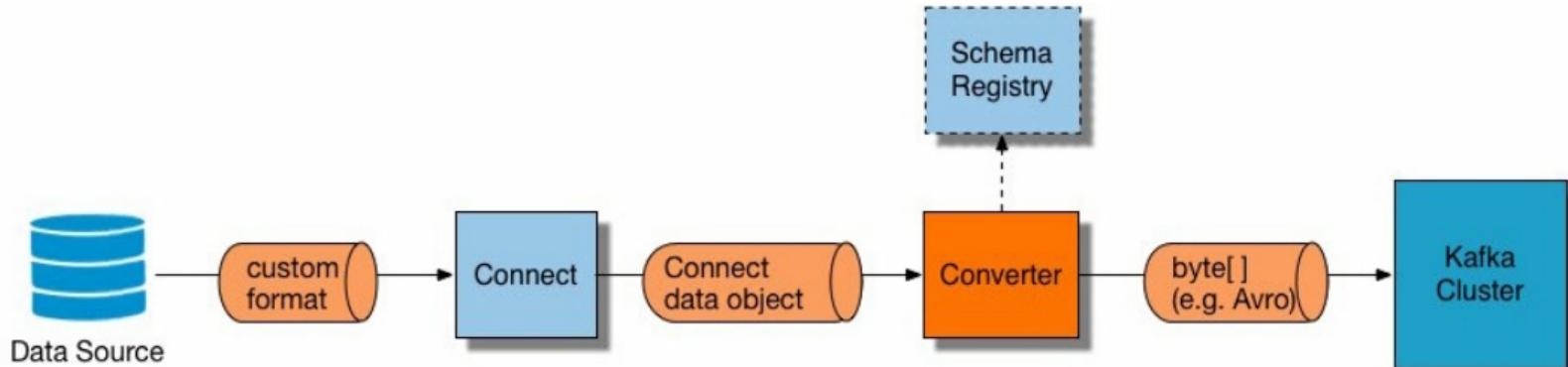
- Splitting the workload into smaller pieces provides the parallelism and scalability
- Connector jobs are broken down into *tasks* that do the actual copying of the data
- *Workers* are processes running one or more tasks, each in a different thread
- Input stream can be partitioned for parallelism, for example:
  - File input: Partition → file
  - Database input: Partition → table



# Converting Data

---

- **Converters provide the data format written to or read from Kafka (like Serializers)**
- **Converters are decoupled from Connectors to enable reuse of converters**
  - Allows any connector to work with any serialization format
- **Example of format conversion in a source converter (sink converter is the reverse)**



# Converter Data Formats

---

- **Converters apply to both the key and value of the message**
  - Key and value converters can be set independently
    - key.converter
    - value.converter
- **Pre-defined data formats for Converter**
  - Avro: AvroConverter
  - JSON: JsonConverter
  - String: StringConverter
  - Byte Array: ByteArrayConverter

# Avro Converter as a Best Practice

---

- **Best Practice is to use an Avro Converter and Schema Registry with the Connectors**

```
key.converter=io.confluent.connect.avro.AvroConverter  
key.converter.schema.registry.url=http://schemaregistry1:8081  
value.converter=io.confluent.connect.avro.AvroConverter  
value.converter.schema.registry.url=http://schemaregistry1:8081
```

# Source and Sink Offsets

---

- **Kafka Connect tracks the produced and consumed offsets so it can restart at the correct place, in case of failure**
- **What the offset corresponds to depends on the Connector, for example:**
  - File input: offset → position in file
  - Database input: offset → timestamp or sequence id
- **The method of tracking the offset depends on the specific Connector**
  - Each Connector can determine its own way of doing this
- **Examples of source and sink offset tracking methods:**
  - JDBC source in distributed mode: a special Kafka Topic
  - HDFS sink: an HDFS file
  - FileStream source: a separate local file

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Types of Connectors*
- *Kafka Connect Implementation*
- **Standalone and Distributed Modes**
- *Configuring the Connectors*
- *Comparing Kafka Connect with Other Options*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

# Two Modes: Standalone and Distributed

---

- **Kafka Connect can be run in two modes**
  - Standalone mode
    - Single worker process on a single machine
    - Use case: testing and development, or when a process should not be distributed (*e.g.* tail a log file)
  - Distributed mode
    - Multiple worker processes on one or more machines
    - Use Case: requirements for fault tolerance and scalability

# Running in Standalone Mode

---

- **To run in standalone mode, start a process by providing as arguments**
  - Standalone configuration properties file
  - One or more connector configuration files
    - Each connector instance will be run in its own thread

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

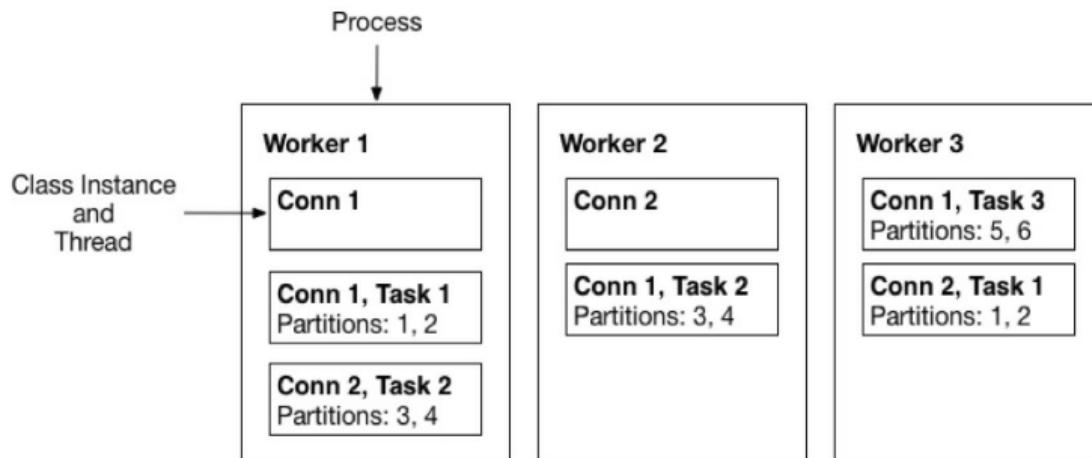
# Running in Distributed Mode

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

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

- Group coordination

- Connect leverages Kafka's group membership protocol
  - Configure workers with the same group.id
  - Workers distribute load within this Kafka Connect “cluster”



# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Types of Connectors*
- *Kafka Connect Implementation*
- *Standalone and Distributed Modes*
- **Configuring the Connectors**
- *Comparing Kafka Connect with Other Options*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

# Transforming Data with Connect

---

- **Kafka 0.10.2 (Confluent 3.2) provides the ability to transform data a message-at-a-time**
  - Configured at the connector-level
  - Applied to the message key or value
- **A subset of available transformations:**
  - InsertField: insert a field using attributes from the message metadata or from a configured static value
  - ReplaceField: rename fields, or apply a blacklist or whitelist to filter
  - ValueToKey: replace the key with a new key formed from a subset of fields in the value payload
  - TimestampRouter: update the Topic field as a function of the original Topic value and timestamp
- **More information on Connect Transformations can be found at**  
**[http://kafka.apache.org/documentation.html#connect\\_transforms](http://kafka.apache.org/documentation.html#connect_transforms)**

# The REST API

---

- Connectors can be added, modified, and deleted via a REST API on port 8083
- In distributed mode, configuration can be done only via this REST API
  - Changes made this way will persist after a worker process restart
  - Connector configuration data is stored in a special Kafka Topic
  - The REST requests can be made to any worker
- In standalone mode, configuration can also be done via a REST API
  - However, typically configuration is done via a properties file
    - Changes made via the REST API when running in standalone mode will not persist after worker restart

# Using the REST API

---

- A subset of the REST API is shown below

Method	Path	Description
<b>GET</b>	<b>/connectors</b>	Get a list of active connectors
<b>POST</b>	<b>/connectors</b>	Create a new Connector
<b>GET</b>	<b>/connectors/(string: name)/config</b>	Get configuration information for a Connector
<b>PUT</b>	<b>/connectors/(string: name)/config</b>	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>

# Configuring Workers: Both Modes

---

- You can modify Connect configuration settings
  - Distributed mode in /etc/kafka/connect-distributed.properties
  - Standalone mode in /etc/kafka/connect-standalone.properties
  - <http://docs.confluent.io/current/connect/>
- Important configuration options common to all workers:

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

- See <http://docs.confluent.io> for a comprehensive list, along with an example configuration file

# Configuring Workers: Standalone Mode

---

- Additional configuration parameter important in standalone mode

Parameter	Description
<b>offset.storage.file.filename</b>	The filename in which to store offset data for the Connectors (Default: ""). This enables a standalone process to be stopped and then resume where it left off.

# Configuring Workers: Distributed Mode

---

- Additional configuration parameters important in 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
<b>offset.storage.topic</b>	The Topic in which to store offset data for the Connectors
<b>status.storage.topic</b>	The Topic in which to store connector and task status
<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>

# Creating Kafka Connect's Topics Manually

---

- The Topics used in Kafka Connect distributed mode will be automatically created
  - They are created according to the defaults for num.partitions (Default: 1) and replication.factor (Default: 1)
  - Instead, consider creating these Topics manually
    - Replication factor set to 3
    - A cleanup.policy set to compact
    - The following number of Partitions:

Topic	Partitions
<b>config.storage.topic</b>	1
<b>offset.storage.topic</b>	50
<b>status.storage.topic</b>	10

# Configuring the Connector

---

- Connect configuration parameters are as follows:

Parameter	Description
<b>name</b>	Connector's unique name
<b>connector.class</b>	Connector's Java class
<b>tasks.max</b>	Maximum tasks to create. The Connector may create fewer if it cannot achieve this level of parallelism (Default: 1)
<b>key.converter</b>	(optional) Override the worker key converter (since Kafka version 0.10.1.0)
<b>value.converter</b>	(optional) Override the worker value converter (since Kafka version 0.10.1.0)
<b>topics</b> (Sink connectors only)	List of input Topics (to consume from)

# Enabling Security with Connect

---

- If you have security enabled in the Kafka cluster (e.g. SSL or SASL), you need to enable it in Connect as well
  - Configure security a per-worker basis
  - Override the default Producer or Consumer configurations that the worker uses
  - Configurations apply to *all* connectors running on the worker

```
# Worker security at the top level
security.protocol=SSL ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
ssl.truststore.password=test1234

# Source security settings are prefixed with "producer"
producer.security.protocol=SSL
producer.ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
producer.ssl.truststore.password=test1234

# Sink security settings are prefixed with "consumer"
consumer.security.protocol=SSL
consumer.ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
consumer.ssl.truststore.password=test1234
```

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Types of Connectors*
- *Kafka Connect Implementation*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- **Comparing Kafka Connect with Other Options**
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

# 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
  - 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
<i>Minimal configuration overhead</i>	Yes	Yes	No	Yes
<i>Support for stream processing</i>	Yes	Lacking	No	Yes
<i>Support for batch processing</i>	Yes	Lacking	Yes	Yes
<i>Focus on reliably and scalably copying data</i>	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><i>Parallel</i></b>	Automated	Manual	Automated	Yes
<b><i>Accessible connector API</i></b>	Yes	Complex	Narrow	Yes
<b><i>Scales across multiple systems</i></b>	Yes	Yes	Yes	Improving

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Types of Connectors*
- *Kafka Connect Implementation*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Comparing Kafka Connect with Other Options*
- **Hands-On Exercise: Running Kafka Connect**
- *Chapter Review*

# Hands-On Exercise: Running Kafka Connect

---

- In this Hands-On Exercise, you will run Connect in distributed mode, interact with the REST proxy, and use a standalone JDBC Source Connector
- Please refer to the Hands-On Exercise Manual

# Kafka Connect for Data Movement

---

- *The Motivation for Kafka Connect*
- *Types of Connectors*
- *Kafka Connect Implementation*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Comparing Kafka Connect with Other Options*
- *Hands-On Exercise: Running Kafka Connect*
- **Chapter Review**

# Chapter Review

---

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

# **Basic Kafka Administration**

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

**>>> 09: Basic Kafka Administration**

10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# Basic Kafka Administration

---

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

- How to setup Kafka components
- How to perform some common Kafka administrative tasks
- How Compacted Logs work
- How to think about the number of Partitions to specify for a Topic
- What security features Kafka provides

# Basic Kafka Administration

---

- **Kafka Versions and Software Upgrades**
- *Administering Kafka*
- *Log Management*
- *Determining How Many Partitions to Specify*
- *Kafka Security*
- *Chapter Review*

# Available Versions

---

- **Confluent provides Kafka in different formats**
  - Available as deb, RPM, Zip archive, tarball
- **Java 7 or Java 8 is required**
- **If you are running on a Mac, use the Zip or Tar archive**
- **Running Kafka on Windows may prove problematic**

# Client and Broker Version Compatibility

---

- **Newer Brokers**

- Clients can communicate with Brokers running newer (or same) versions of Kafka
    - Example: a Kafka 0.10.0 (Confluent 3.0) Producer can communicate with a Kafka 0.10.1 (Confluent 3.1) Broker

- **Older Brokers**

- Clients can communicate with Brokers running older versions of Kafka
    - Example: a Kafka 0.10.2 (Confluent 3.2) Producer can communicate with a Kafka 0.10.1 (Confluent 3.1) Broker
  - Caveats
    - Clients must be running at least Kafka 0.10.2 (Confluent 3.2)
    - Brokers must be running at least Kafka 0.10.0 (Confluent 3.0)
    - Some features may not be available

# Upgrading Kafka

---

- **Typically, a Kafka cluster does not need to be completely shut down in order to be upgraded**
  - Instead, you can usually perform a rolling upgrade
- **Upgrade the Brokers before upgrading clients**
- **Check the documentation for full details!**

# Basic Kafka Administration

---

- *Kafka Versions and Software Upgrades*
- **Administering Kafka**
- *Log Management*
- *Determining How Many Partitions to Specify*
- *Kafka Security*
- *Chapter Review*

# Introduction

---

- Note that we only cover a few common administrative functions here
- For much more in-depth coverage, consider attending *Confluent Operations Training for Kafka*

# Configuring Topics

---

- **By default, Topics are automatically created when they are first used by a client**
  - Auto-created Topics will have a single Partition and a single replica, by default
  - 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 my_topic \
--partitions 6 --replication-factor 3
```

- **You can also modify Topics**

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

- **No data is moved from existing Topics**
- **Changing the number of Partitions could cause problems for your application logic!**

# Deleting Topics

---

- Since Kafka 1.0 (Confluent 4.0), Topic deletion is enabled by default on Brokers
  - delete.topic.enable (Default: true)
- Caveats
  - Stop all Producers/Consumers before deleting
  - All Brokers must be running for the delete to be successful

```
$ kafka-topics --zookeeper zk_host:2181 --delete --topic my_topic
```

# Basic Kafka Administration

---

- *Kafka Versions and Software Upgrades*
- *Administering Kafka*
- **Log Management**
- *Determining How Many Partitions to Specify*
- *Kafka Security*
- *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 Compaction (1)

---

- **When cleaning up a log, the default policy is delete**
  - Removes all log segments where all messages are older than the retention.ms or until the log size is below retention.bytes (or both)
- **An alternate policy is compact**
  - A compacted log retains at least the last known message value for each key within the Partition
  - Example usage scenarios
    - A database change log
    - Storing state for external applications
- **Configure the retention policy with the Broker-level log.cleanup.policy configuration parameter**
  - Use the cleanup.policy configuration parameter to override at the Topic level
- **As of Kafka 0.10.1 (Confluent 3.1), policy supports both delete and compact to be enabled at the same time**

## Log Compaction (2)

Offset	0	1	2	3	4	5	6	7	8	9	10
Key	K1	K2	K1	K1	K3	K2	K4	K5	K5	K2	K6
Value	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11

Compaction

Keys	3	4	6	8	9	10
	K1	K3	K4	K5	K2	K6
Values	V4	V5	V7	V9	V10	V11

Log  
After  
Compaction

# Log Compaction (3)

---

- **It is possible that multiple values may exist in the log for a given key**
  - If new values are written after the most recent cleanup
- **Clients should read the entire log to ensure they have the latest value for each key**
- **You can configure how aggressive log compaction is**
  - `log.cleaner.min.cleanable.ratio`
    - Ratio of “dirty” (uncompacted) log to total log size (excluding the currently active log segment)
    - Trigger log clean if the ratio of dirty/total is larger than this value
    - Default is 0.5
    - A Topic can override a Broker’s configured ratio with the property `min.cleanable.dirty.ratio`
- **Question: if a Consumer offset points to a message that has been compacted away, does `auto.offset.reset` apply?**

# Basic Kafka Administration

---

- *Kafka Versions and Software Upgrades*
- *Administering Kafka*
- *Log Management*
- **Determining How Many Partitions to Specify**
- *Kafka Security*
- *Chapter Review*

# Specifying the Number of Partitions

---

- If a Topic is automatically created, it will have a single Partition and a single replica, by default
- If a Topic is manually created, you can specify the number of Partitions, and the number of replicas for each Partition

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

- The number of Partitions can have an impact on performance

# More Partitions → Higher Throughput

---

- **Parallelization**

- Writes to Partitions by the Producer can be performed in parallel
    - Therefore, more Partitions can result in better throughput when sending data to the cluster
  - Reads from Topics by a Consumer Group are bounded by the number of Partitions
    - If you have more Consumers in a Consumer Group than you have Partitions in a Topic, some Consumers will receive no data for that Topic

# The Downside of Many Partitions

---

- **Partition unavailability in the event of Broker failure**
  - If a Broker fails, and it is a leader for some number of Partitions, each of those Partitions will become unavailable until a new leader is elected
    - The more Partitions for which a Broker is a leader, the longer this will take
- **End-to-end latency (time from Producing a message to it being read by a Consumer)**
  - A Broker uses a single thread to replicate data from another Broker with which it shares Partitions
    - The more Partitions there are, the longer this can take
  - Alleviated on larger clusters where replicas are distributed amongst more Brokers
- **Memory requirements (buffers on clients are per Partition)**
  - More Partitions can lead to larger memory requirements on the client
    - The limit on how much data is sent back to the client is a per-Partition limit
    - Producers may locally cache messages for batching purposes
  - Mitigated with max.poll.records to reduce the memory requirements

# Basic Kafka Administration

---

- *Kafka Versions and Software Upgrades*
- *Administering Kafka*
- *Log Management*
- *Determining How Many Partitions to Specify*
- **Kafka Security**
- *Chapter Review*

# Encryption, Authentication, Authorization

---

- **Kafka provides encryption, authentication, and authorization**
  - Authentication: “Prove you are who you say you are”
  - Authorization: “Here is what you are allowed to do”

# Encryption and Authentication

---

- **Kafka Brokers can listen on multiple ports:**
  - Plain text (no wire encryption or authentication)
  - SSL (wire encryption and optional SSL authentication)
  - SASL: Simple Authentication and Security Layer
    - GSSAPI: Kerberos, introduced in Kafka 0.9.0 (Confluent 2.0)
    - SCRAM-SHA-256, SCRAM-SHA-512: “salted” and hashed passwords, introduced in Kafka 0.10.2 (Confluent 3.2)
  - PLAIN: cleartext username/password, introduced in Kafka 0.10.0 (Confluent 3.0)
- **Clients choose which ports to use**
  - Required credentials are provided through configuration
- **Note: encryption is across the wire only**
  - No encryption at rest

# SSL Performance Impact

---

- **Performance was measured on Amazon EC2 r3.xlarge instances**
  - Note that these performance tests were not run with Java 9, which has significant performance improvement

	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%

# Authorization Using ACLs

---

- **Kafka supports authorization using Access Control Lists (ACLs)**
- **Common use cases:**
  - Allow Producers to write to certain Topics
  - Allow Consumers to read from certain Topics
  - Allow certain users to issue a controlled shutdown
- **ACLs are enabled in the Broker's configuration file**
- **ACLs are configured via the kafka-acls command-line tool**
  - Example: allow user Bob to write to and read from Topic my\_topic

```
$ bin/kafka-acls --authorizer-properties \
zookeeper.connect=zookeeper1:2181 --add \
--allow-principal User:Bob --operation Read --operation Write \
--topic my_topic
```

# Securing The Schema Registry

---

- The Schema Registry can be configured with security
- Secure communication and authentication between the Schema Registry and the Kafka cluster
  - SSL
  - SASL
- Secure communication for end-user REST API calls (HTTPS)
  - SSL
- Authentication with ZooKeeper
  - SASL
- Since Confluent Platform 4.0: Schema Registry security plugin restrict schema evolution to administrative users
  - Client application users get read-only access only
- Details at <http://docs.confluent.io/current/schema-registry/docs/config.html>

# Securing The REST Proxy

---

- As of Kafka 0.10.2 (Confluent 3.2), the REST Proxy can be configured with security
- Secure communication between REST clients and the REST Proxy (HTTPS)
  - SSL
  - Since Confluent Platform 4.0: REST proxy security plugin propagates client principal authentication to Kafka brokers
- Secure communication between the REST Proxy and Apache Kafka
  - SSL
  - SASL
- Authentication with ZooKeeper
  - SASL
  - Client application users get read-only access only
- Details at <http://docs.confluent.io/current/kafka-rest/docs/config.html>

# Basic Kafka Administration

---

- *Kafka Versions and Software Upgrades*
- *Administering Kafka*
- *Log Management*
- *Determining How Many Partitions to Specify*
- *Kafka Security*
- **Chapter Review**

# Chapter Review

---

- Topics can be manually created, modified and deleted
- Compacted logs are useful in some scenarios
- Increasing the number of Partitions for a Topic can improve performance in some situations
- Kafka 0.9 and above supports encryption, authentication, and authorization

# Kafka Stream Processing

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

>>> 10: Kafka Stream Processing

11: Conclusion

Appendix A: Installation Recommendations

# Kafka Stream Prcoessing

---

- In this chapter you will learn:

- The motivation behind the Streams API in Kafka
- The features provided by the Streams API
- How to write an application using the Streams DSL (Domain-Specific Language)
- How to use KSQL
- Resource demands for Kafka Streams and KSQL

# Kafka Stream Processing

---

- **An Introduction to the Kafka Streams API**
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# What Is the Kafka Streams API?

---

- **The Kafka Streams API transforms and enriches data**
  - Supports per-record stream processing with millisecond latency
  - Supports stateless processing, stateful processing, windowing operations
- **Kafka Streams API has fault-tolerance and supports distributed processing**
- **Kafka Streams API has a Domain-Specific Language (DSL)**
  - Provides the most common data transformation operations such as map, flatMap, count, etc.
- **The Kafka Streams API is part of the open-source Apache Kafka project**
- **There are community-driven implementations of the Kafka Streams API in other languages (e.g., Python, Node.js)**

# A Library, Not a Framework

---

- **The Kafka Streams API provides streaming capabilities**
- **There are other Streaming frameworks**
  - Spark Streaming
  - Apache Storm
  - Apache Samza
  - etc.
- **Unlike these, the Streams API in Kafka does not require its own cluster**
  - The Streams API is just a library in Kafka
  - Can run on a stand-alone machine, or multiple machines
- **Using the Streams API in Kafka may reduce data and load on remote database management systems**
  - Local data accesses via the local persistent datastore

# 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 API 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

# Streams API and Broker Version Compatibility

---

- **Newer Brokers**

- Kafka Streams API can communicate with Brokers running newer (or same) versions of Kafka
    - Example: Streams API 0.10.0 (Confluent 3.0) can communicate with a Kafka 0.10.2 (Confluent 3.2) Broker

- **Older Brokers**

- Starting in Kafka 0.10.2 (Confluent 3.2), Streams API 0.10.2 needs Brokers to be running at least Kafka 0.10.1 (Confluent 3.1)
    - Example: Streams API 0.10.2 (Confluent 3.2) can communicate with a Kafka 0.10.1 (Confluent 3.1) Broker

- **The Streams API is not compatible with clusters running older Kafka Brokers (0.7, 0.8, 0.9)**

# Kafka Stream Processing

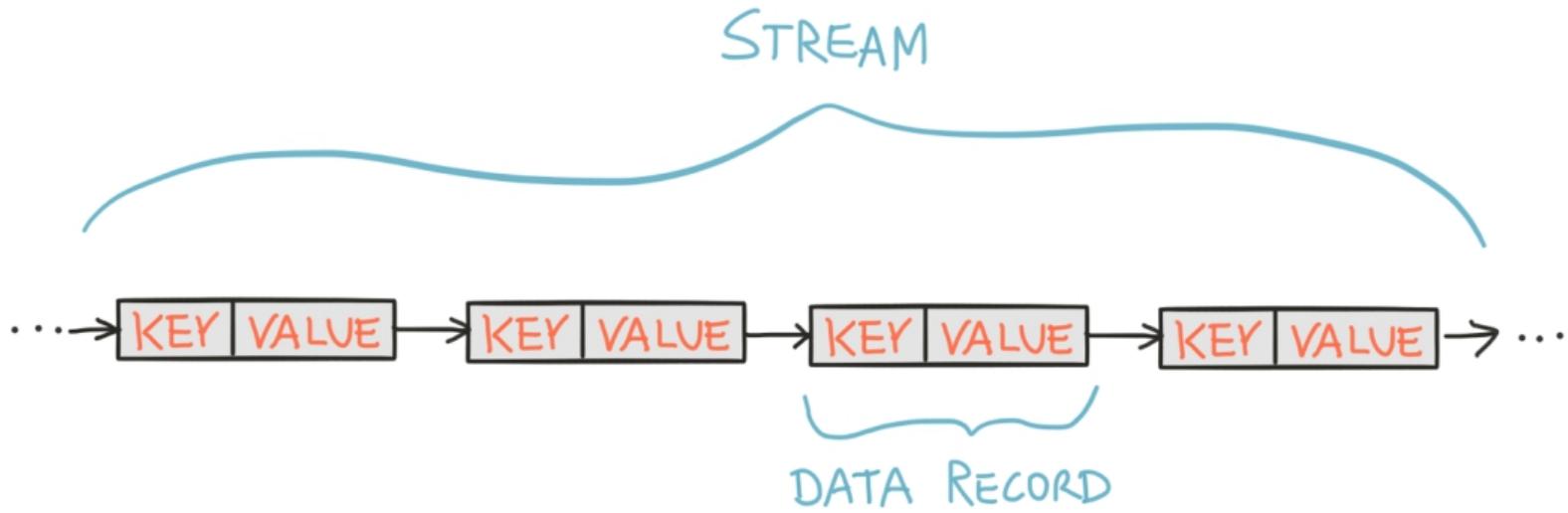
---

- *An Introduction to the Kafka Streams API*
- **Kafka Streams Concepts**
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# What is a Stream?

---

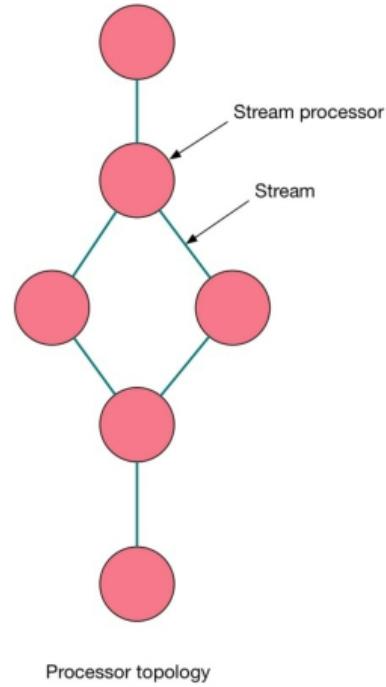
- **Think of a stream as an unbounded, continuous real-time flow of records**
  - You don't need to explicitly request new records, you just receive them
- **Records are key-value pairs**



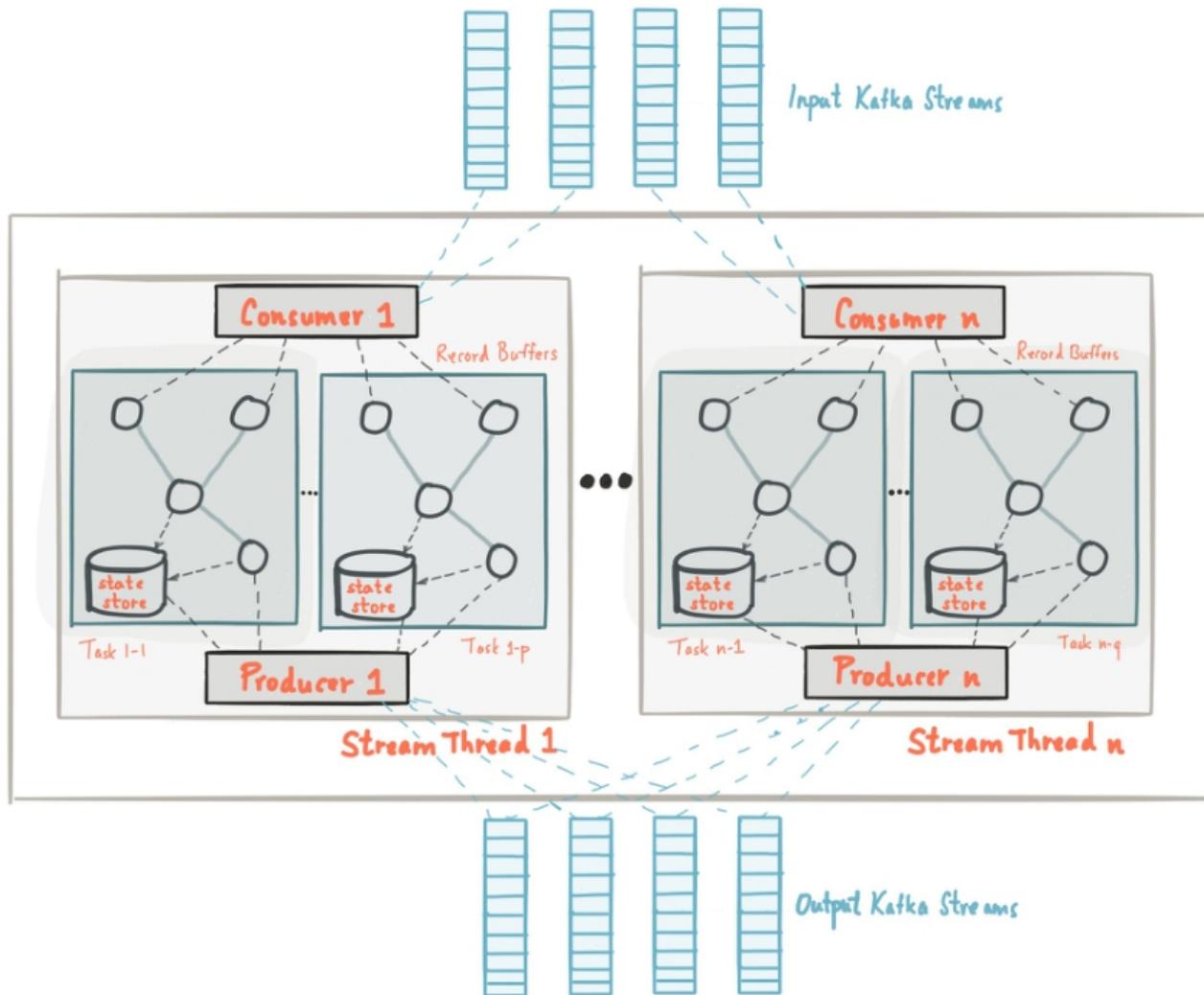
# Stream Processor and Topology

---

- A *stream processor* transforms data in a stream
- A *processor topology* defines the data flow through the stream processors



# Logical View of Kafka Streams Application



# 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, Joining, and Aggregations

---

- **Kafka Streams API allows us to *window* the stream of data by time**
  - To divide it up into ‘time buckets’
- **We can aggregate records**
  - Combine multiple input records together in some way into a single output record
  - Examples: sum, count
  - This is usually done on a windowed basis
- **We can join, or *i.e.*, merge, data from different sources**

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- **Creating a Kafka Streams Application**
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# Configuring the Application

---

- **Kafka Streams configuration is specified with a StreamsConfig instance**
  - This is typically created from a java.util.Properties instance
- **Specify configuration parameters**
  - APPLICATION\_ID\_CONFIG: app name must be unique in the Kafka cluster
  - BOOTSTRAP\_SERVERS\_CONFIG: where to find Kafka broker(s)
- **Example:**

```
Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-example");
streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
streamsConfiguration.put(..., ...); ①
```

- ① Continue to specify configuration options

# Serializers and Deserializers (Serdes)

---

- Use **Serializers and Deserializers (Serdes)** to convert bytes of the record to a specific type
  - SERializer
  - DESerializer
- Key Serdes can be independent from value Serdes
- There are many many built-in Serdes (e.g. Serdes.String, etc.)
- You can also define your own
- Configuration example:

```
streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass());
```

# Available Serdes

---

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

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

- If you are using Avro, you can use Avro Serde provided by the kafka-streams-avro-serde Maven artifact

# Creating the Processing Topology

---

- Create a KStream or KTable object from one or more Kafka Topics, using StreamsBuilder
- Example:

```
StreamsBuilder builder = new StreamsBuilder();
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

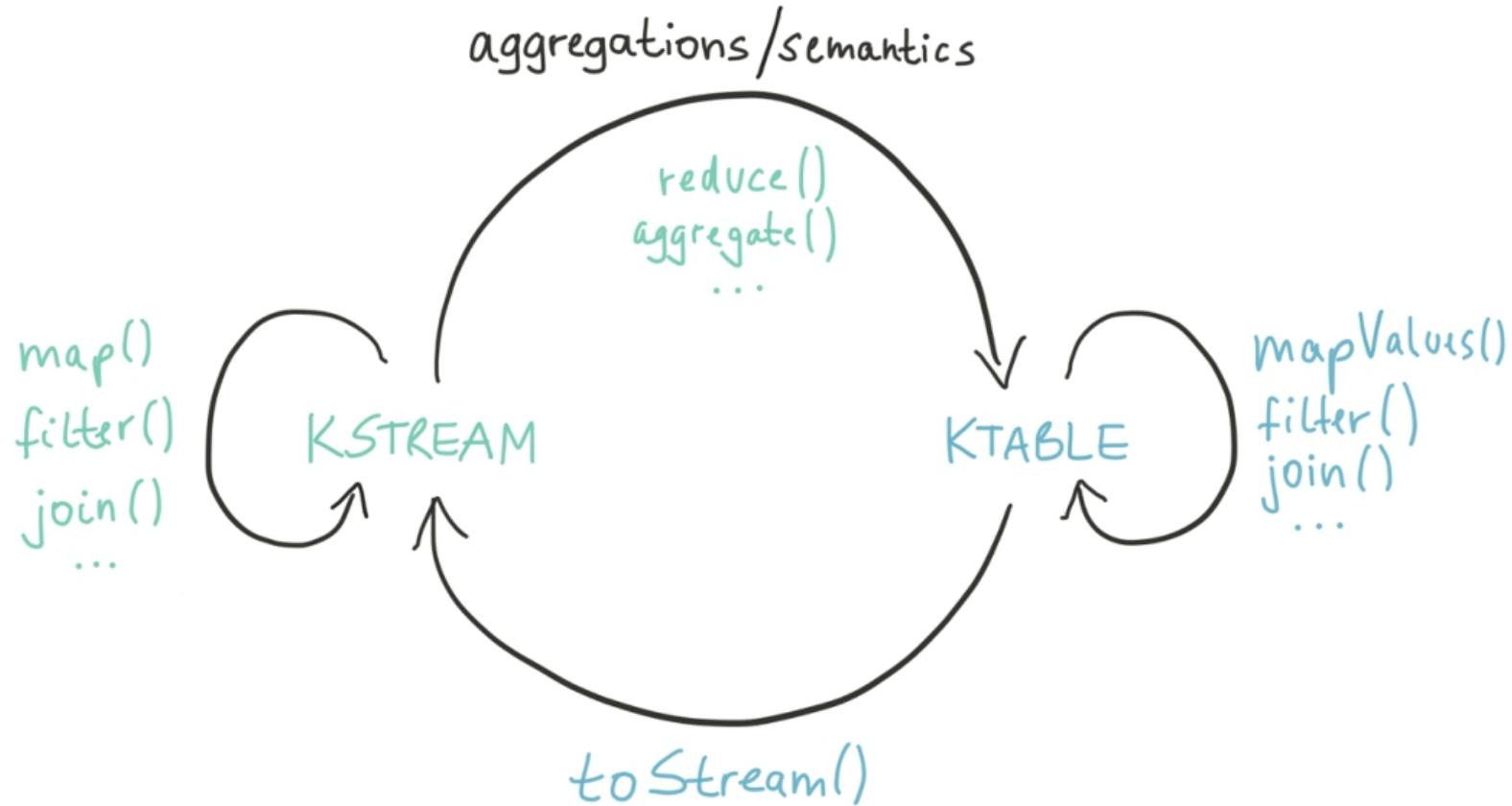
---

- **Examples of stateful transformation operations:**
  - `stream.groupByKey().count()`
    - Counts the number of instances of each key in the stream; results in a new, ever-updating KTable
  - `stream.groupByKey().reduce()`
    - 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/current/streams/javadocs/index.html>**

# Mixed Processing

---

- You can build stream processing topologies with mixed stateless and stateful processing



# 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
- **Use print() to write the contents of the stream**

```
myNewStream.print(Printed.toSysOut());
```

# Running the Application

---

- To start processing the stream, create a **KafkaStreams** object
  - Configure it using StreamsBuilder
- Then call the **start()** method
- Example:

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

# Application Graceful Shutdown

---

- **Allow your application to gracefully shutdown**
  - For example, in response to SIGTERM
- **Add a shutdown hook to stop the application**

```
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

- **After an application is stopped, Kafka migrates any tasks that had been running in this instance to available remaining instances**

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- **Kafka Streams By Example**
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# A Simple Kafka Streams API Example (1)

---

- Define the Kafka Streams configuration properties

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

# A Simple Kafka Streams Example (2)

---

- Create the processing topology, transform the data, run the application

```
12  StreamsBuilder builder = new StreamsBuilder();
13
14  // Construct a KStream from the input Topic "TextLinesTopic"
15  KStream<byte[], String> textLines = builder.stream("TextLinesTopic");
16
17  // Convert to upper case (:: is Java 8 syntax)
18  KStream<byte[], String> uppercasedWithMapValues = textLines.mapValues(String::toUpperCase);
19
20  // Write the results to a new Kafka Topic called "UppercasedTextLinesTopic".
21  uppercasedWithMapValues.to("UppercasedTextLinesTopic");
22
23  // Run the Streams application via `start()`
24  KafkaStreams streams = new KafkaStreams(builder.build(), streamsConfiguration);
25  streams.start();
26
27  // Stop the application gracefully
28  Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
29 }
30 }
```

# Kafka Streams With Stateful Processing

```
1  StreamsBuilder builder = new StreamsBuilder();
2
3  // Construct a KStream from the input Topic "TextLinesTopic"
4  KStream<String, String> textLines = builder.stream("TextLinesTopic");
5
6  final KTable<String, Long> wordCounts = textLines
7      .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
8      // Count the occurrences of each word (record key).
9      .groupBy((key, word) -> word)
10     .count("Counts");
11
12 // Write the `KTable<String, Long>` to an output Topic
13 wordCounts.toStream().to("WordsWithCountsTopic", Produced.with(Serdes.String(), Serdes.
14 Long()));
15
16 // Run the Streams application via `start()`.
17 KafkaStreams streams = new KafkaStreams(builder.build(), streamsConfiguration);
18 streams.start();
19
20 Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
21 }
```

# Join Example

---

- You can enrich records by merging data, *i.e.*, joining data, from different sources
- The example below does a join based on key
  - Effectively, where `leftStream.key == rightStream.key`, use the value from `leftStream`

```
1 final KStream<Long, Long> joinedStream =  
2     leftStream.leftJoin(rightStream, (leftValue, rightValue) -> leftValue, Serdes.Long(), Serdes  
.Long());
```

# Using Avro with Kafka Streams API

---

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

# Converting Our Avro Example to Kafka Streams

```
1 import solution.model.ShakespeareKey;
2 import solution.model.ShakespeareValue;
3
4 // Specify Avro Serde and Schema Registry
5 streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, SpecificAvroSerde.class);
6 streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, SpecificAvroSerde.
class);
7 streamsConfiguration.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://schemaregistry1:8081");
8
9 StreamsBuilder builder = new StreamsBuilder();
10
11 KStream<String, String> textLines = builder.stream("shakespeare_topic",
Consumed.with(Serdes.String(), Serdes.String()));
12 KStream<ShakespeareKey, ShakespeareValue> converted = textLines
13 .map((k, line) -> new KeyValue<ShakespeareKey, ShakespeareValue>(getShakespeareKey(k),
getShakespeareLine(line))); ①
14
15 converted.to("streaming_shakespeare_output");
16
17 // Continue with stream builder, start, and shutdown hook
```

- ① These are the methods you created in the Exercise

# 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 (EOS) is supported in Kafka 0.11 (Confluent 3.3)**
  - In your Kafka Streams application, set `processing.guarantee` to `exactly_once`  
(Default: `at_least_once`)

# More Kafka Streams API 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...

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- **Hands-On Exercise: Writing a Kafka Streams Application**
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# Hands-On Exercise: Writing a Kafka Streams Application

---

- In this Hands-On Exercise, you will write an application which uses the Streams API in Kafka to process data
- Please refer to the Hands-On Exercise Manual

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- **Managing Kafka Stream Processing**
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# Parallelizing Kafka Streams

---

- You can run Kafka streaming applications across multiple machines
- Parallelizing Kafka Streams or KSQL applications can improve throughput and performance
- Kafka handles dividing the work across the machines
- It also provides fault tolerance
  - If a machine fails, the task it was working on is automatically restarted on one of the remaining instances
- The number of tasks in the Kafka Streams application may be increased by increasing the number of Partitions

# Fault Tolerance

---

- **Clients can enable standby replica tasks for the streaming applications**
  - These standby replica tasks have fully replicated copies of the state
  - When a task migration happens, Kafka Streams attempts to assign a task to an application instance where such a standby replica already exists
  - Reduces restoration time for a failed task
- **num.standby.replicas: number of standby replicas for each task (Default: 0)**

# Client Resource Utilization

---

- **Stateful client applications may use Streams operations (e.g., aggregates, joins, windows)**
- **Clients keep state in local state stores (*i.e.*, RocksDB)**
  - The state stores use local disk on the client machines
  - The state stores have 50MB - 100MB memory overhead
  - Clients persist state stores to a compacted Kafka Topic
    - If the client state store fails, the data is recoverable
  - Using standby replicas for Kafka Streams tasks will result in extra copies of state stores
- **If client performance is bound by CPU**
  - Add cores or machines, and increase the number of threads
    - num.stream.threads (Default: 1)
- **If client performance is bound by network, memory, or disk**
  - Scale out the clients on to additional machines

# Capacity Planning: Brokers

---

- **How clients use stream processing may impact utilization of the Kafka cluster**
  - Kafka Streams API persists the local client state store to the Kafka cluster
- **The more heavily the local client state store is used, the more data gets written to Kafka**
  - This may require increased capacity in the cluster
- **Using standby replicas for Kafka Streams tasks will result in extra Consumers and network bandwidth utilization**

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- **KSQL for Apache Kafka**
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# Using KSQL to Query and Enrich Data

---

- **KSQL is the streaming SQL engine for Apache Kafka**
  - KSQL transforms and enriches data in a Kafka cluster
  - Does real-time stream processing using the Kafka Streams API underneath
- **KSQL is an alternative to writing a Java application**
- **KSQL STREAM and TABLE abstractions are analogous to Kafka Streams API KStream and KTable**
- **KSQL is downloadable from GitHub**
  - <https://github.com/confluentinc/ksql>

# SQL-like Semantics

---

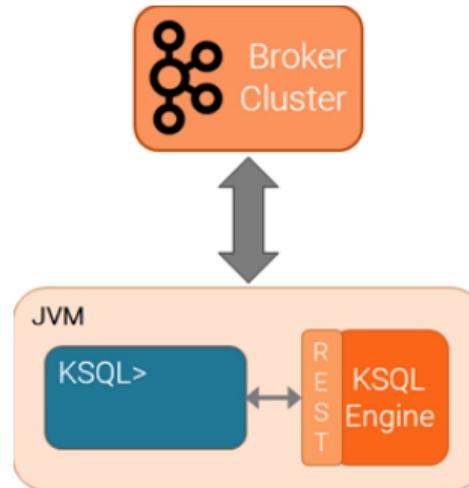
- KSQL lets users read, write, and process streaming data using SQL-like semantics
- Interface should be familiar to users of MySQL, Postgres, Oracle, Hive, Presto, etc.
- Here is a KSQL example:

```
CREATE TABLE possible_fraud AS
  SELECT card_number, count(*)
  FROM authorization_attempts
  WINDOW TUMBLING (SIZE 5 SECONDS)
  GROUP BY card_number
  HAVING count(*) > 3;
```

# Main Components of KSQL

---

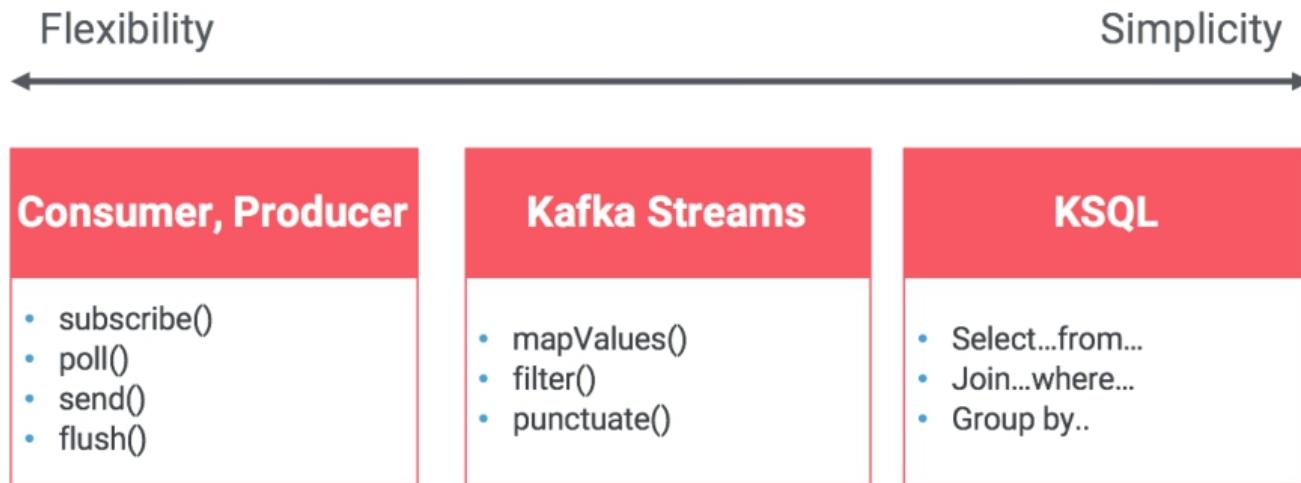
- **The main components of KSQL are the KSQL server and the KSQL CLI**
  - KSQL server runs the engine that executes KSQL queries
    - Processes data, and writes to and reads from the Kafka cluster
  - KSQL CLI acts as a client to the KSQL server
    - Allows you to interactively write KSQL queries
- **Run in standalone mode, or client-server mode with a pool of KSQL servers**



# Comparing KSQL to Kafka Streams to Producer/Consumer

---

- KSQL is an API around Kafka Streams, and Kafka Streams is an API around Producer and Consumer
- Choose the right API for your business



# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- **Writing KSQL Queries**
- *Hands-On Exercise: Data Transformations with KSQL*
- *Chapter Review*

# KSQL Syntax

---

- **KSQL has similar semantics to SQL**
- **KSQL statements must be terminated with a semicolon ;**
- **Multi-line statements**
  - Use a back-slash \ to indicate continuation of a statement on the next line

# General KSQL Commands

Command	Description
<i>DESCRIBE</i>	List the columns in a stream or table along with their data type and other attributes
<i>CREATE STREAM</i>	Create a new stream with the specified columns and properties
<i>CREATE TABLE</i>	Create a new table with the specified columns and properties
<i>SHOW TOPICS</i>	List the available Topics in the Kafka cluster that KSQL is configured to connect to
<i>SHOW STREAMS</i>	List the defined streams
<i>SHOW TABLES</i>	List the defined tables
<i>DROP</i>	Drop an existing stream or table

# Non-persistent and Persistent Queries

---

- **Non-persistent query**
  - The result of a non-persistent query will not be persisted into a Kafka Topic and will only be printed out in the console
- **Persistent query**
  - The result of a persistent query will be persisted into a Kafka Topic

Command	Description
<i>SELECT</i>	Non-persistent
<i>CREATE STREAM AS SELECT</i>	Persistent
<i>CREATE TABLE AS SELECT</i>	Persistent

# Stopping Queries

---

- **By default, SELECT won't stop on its own**
  - Since these are never-ending *streams* of data, there is no inherent “end”
- **There are differences in stopping a non-persistent versus persistent query**
  - Stop a non-persistent query with ctrl-c to the console
  - Stop a persistent query with the TERMINATE command
- **You may also limit the number of records returned with the LIMIT clause**

# Explore Data in Kafka Topics

---

- View the content of the Kafka topic **my-pageviews-topic**, which has a JSON payload, in the form of a stream

```
CREATE STREAM pageviews (viewtime BIGINT, user_id VARCHAR, page_id VARCHAR)
    WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'my-pageviews-topic');
```

- View the content of the Kafka topic **my-users-topic**, which has a JSON payload, in the form of a changelog

```
CREATE TABLE users (usertimestamp BIGINT, user_id VARCHAR, gender VARCHAR, region_id VARCHAR)
    WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'my-users-topic');
```

# KSQL Functions and Clauses

---

- KSQL supports many popular scalar functions that have similar meaning to its counterpart in SQL
  - ABS, CEIL, CONCAT, LEN, ROUND, TRIM, SUBSTRING, and many more
  - EXTRACTJSONFIELD: Given a string column in JSON format, extract the field that matches
- You can JOIN streams and tables
- KSQL supports aggregation functions as well
  - COUNT, MAX, MIN, SUM, etc
- Re-key the streams
  - With the PARTITION BY clause, the resulting stream will have a key with the specified column

# Persistent Query With Join Example

---

- Create a persistent query that does a **JOIN** between a stream and table based on a field **userid**

```
CREATE STREAM pageviews_female AS
  SELECT users.userid AS userid, pageid, regionid, gender
  FROM pageviews
  LEFT JOIN users
  ON pageviews.userid = users.userid
  WHERE gender = 'FEMALE';
```

# Windows in KSQL

---

- **Use the WINDOW clause to group input records for aggregations or joins into**
  - TUMBLING: fixed-sized, non-overlapping windows based on the records' timestamps
  - HOPPING: fixed-sized, (possibly) overlapping windows based on the records' timestamps
  - SESSION: sessions with a period of activity and gaps in between
- **Specify window size, *i.e.*, period of time**

# Persistent Query With Windowing Example

---

- Create a persistent query that counts data in a tumbling window.

```
CREATE TABLE pageviews_regions AS
    SELECT gender, regionid , COUNT(*) AS numusers
    FROM pageviews_female
    WINDOW TUMBLING (size 30 second)
    GROUP BY gender, regionid
    HAVING COUNT(*) > 1;
```

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- **Hands-On Exercise: Data Transformations with KSQL**
- *Chapter Review*

# Hands-On Exercise: Data Transformations with KSQL

---

- In this Hands-On Exercise, you will use KSQL to transform data and identify patterns
- Please refer to the Hands-On Exercise Manual

# Kafka Stream Processing

---

- *An Introduction to the Kafka Streams API*
- *Kafka Streams Concepts*
- *Creating a Kafka Streams Application*
- *Kafka Streams By Example*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Managing Kafka Stream Processing*
- *KSQL for Apache Kafka*
- *Writing KSQL Queries*
- *Hands-On Exercise: Data Transformations with KSQL*
- **Chapter Review**

# Chapter Review

---

- **Kafka Streams API provides a DSL for writing Kafka stream processing applications in Java**
  - It is a lightweight library
  - No external dependencies other than core Kafka
- **No external cluster is required, unlike most stream processing frameworks**
- **KSQL is a declarative stream processing language built on top of Kafka Streams API**

# Summary

# Course Contents

---

01: Introduction

02: The Motivation for Apache Kafka

03: Kafka Fundamentals

04: Kafka's Architecture

05: Developing With Kafka

06: More Advanced Kafka Development

07: Schema Management In Kafka

08: Kafka Connect for Data Movement

09: Basic Kafka Administration

10: Kafka Stream Processing

>>> 11: Conclusion

Appendix A: Installation Recommendations

# Conclusion

---

- **During this course, you have learned:**

- The motivation for Apache Kafka
- 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
- How the REST Proxy supports development in languages other than Java
- Common patterns for application development
- How to integrate Kafka with external systems using Kafka Connect
- Basic Kafka cluster administration
- How to write streaming applications with the Kafka Streams API and KSQL

# 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)**

# Installation Recommendations

# Installation Recommendations

---

- **Kafka Installation**
- *Hardware Considerations*
- *Chapter Review*

# Operating System and Software Requirements

---

- Choose the Linux server operating system you are most familiar with
  - Red Hat Enterprise Linux/CentOS and Ubuntu are the most common options
- We recommend the latest release of JDK 1.8
- Use the G1 Garbage Collector
- Typical JVM options:

```
-Xms4g -Xmx4g -XX:MetaspaceSize=96m -XX:+UseG1GC -XX:MaxGCPauseMillis=20  
-XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M -  
XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
```

# File System Configuration

---

- **File system**
  - Use XFS or ext4
  - Mount with noatime

# Increase Open File Handle Limit

---

- **Increase the open file handle limit**
  - Kafka needs a file descriptor for each socket, log segment, index segment, and timeindex segment

```
$ ulimit -n 100000
```

# Installing ZooKeeper

---

- **Confluent's distribution includes ZooKeeper 3.4.10**
- **In production, you will typically run a ZooKeeper quorum of 3 or 5 machines**
- **ZooKeeper is sensitive to I/O latency**
  - If the ZooKeeper nodes run other processes (*e.g.*, Kafka Brokers), make sure that ZooKeeper has its own disk

# Configuring Brokers

---

- Confluent's distribution comes with sample configuration files for the Broker, ZooKeeper, Kafka Connect etc.
- Each Broker must have its own unique ID
  - An integer, set in the broker.id property
- The Broker takes its hostname from the value returned by `java.net.InetAddress's getCanonicalHostName() method`
  - If this is incorrect, set the advertised.listeners property

# The Schema Registry

---

- **The Schema Registry typically resides on its own server**
- **It stores schema information in a Kafka Topic, determined by the `kafkastore.topic` configuration value**
  - By default this is a Topic named `_schemas`
- **The Schema Registry requires access to the ZooKeeper quorum**
  - Identify the ZooKeeper connection in `kafkastore.connection.url`

# The REST Proxy

---

- If you intend to use the REST Proxy you can place it on a server running a Kafka Broker, or on a stand-alone server (recommended for production)
- For heavy workloads, multiple instances can be placed behind a load balancer
- Each REST Proxy should have its own unique ID number, in the id configuration parameter
- The REST Proxy requires access to the ZooKeeper quorum
  - List all ZooKeeper instances in zookeeper.connect
- It also requires access to a running Schema Registry
  - Place its location in schema.registry.uri

# Installation Recommendations

---

- *Kafka Installation*
- **Hardware Considerations**
- *Chapter Review*

# Broker System Requirements

---

- **Critical resources for a Kafka Broker:**
  - Disk space and I/O
  - Network bandwidth
  - RAM (for page cache)

# Broker Disk

---

- For lower latency, always use local disk instead of shared storage like NAS or SAN
- Use RAID-10 rather than RAID-5 or RAID-6
  - Hides single disk failure
  - Check the performance of a hot disk swap
- What about JBOD (Just a Bunch Of Disks)?
  - Provides more capacity (no RAID overhead)
  - The Broker stores any given Partition of a Topic on a single volume
  - Since Kafka 1.0 (Confluent 4.0) a single disk failure will no longer bring down the entire Broker
    - The Broker will continue serving any log files that remain on functioning disks
    - Potentially uneven distribution of data

# Network Considerations

---

- **Gigabit Ethernet is sufficient for many applications**
  - 10Gb Ethernet will help for large installations
  - Particularly for inter-Broker communication

# Server RAM Specification

---

- **Servers do not need very large amounts of RAM**
- **Kafka Brokers themselves have a relatively small memory footprint**
- **Extra RAM will be used by the operating system for disk caching**
  - This is the desired behavior for a Kafka Broker
- **When specifying CPUs, favor more cores over faster cores**
  - Kafka is heavily multi-threaded

# Installation Recommendations

---

- *Kafka Installation*
- *Hardware Considerations*
- **Chapter Review**

# Chapter Review

---

- Recommendations for Kafka installation
- Hardware considerations