confluent

Confluent Developer Training: Building Kafka Solutions

B1/604/A

Introduction

Chapter 01

Course Contents

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 Hadoop using Kafka Connect
  - Basic Kafka cluster administration
  - The basic features of Kafka Streams
- **Throughout the course, Hands-On Exercises will reinforce the topics being discussed**

Introduction

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

About Kafka

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

About Confluent

- **Founded in 2014 by the creators of Kafka**
- **Provides world-class support, consulting, and training services**
- **Provides the Confluent Platform**
  - Kafka plus critical bugfixes not yet applied to the open source release
  - Kafka ecosystem projects
    - Example: Kafka Connect
  - Enterprise support

Introduction

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

Class Logistics

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

Course Materials

- **Your instructor will give you details on how to access the course materials**

Introductions

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

The Motivation for Apache Kafka

Chapter 02

Course Contents

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

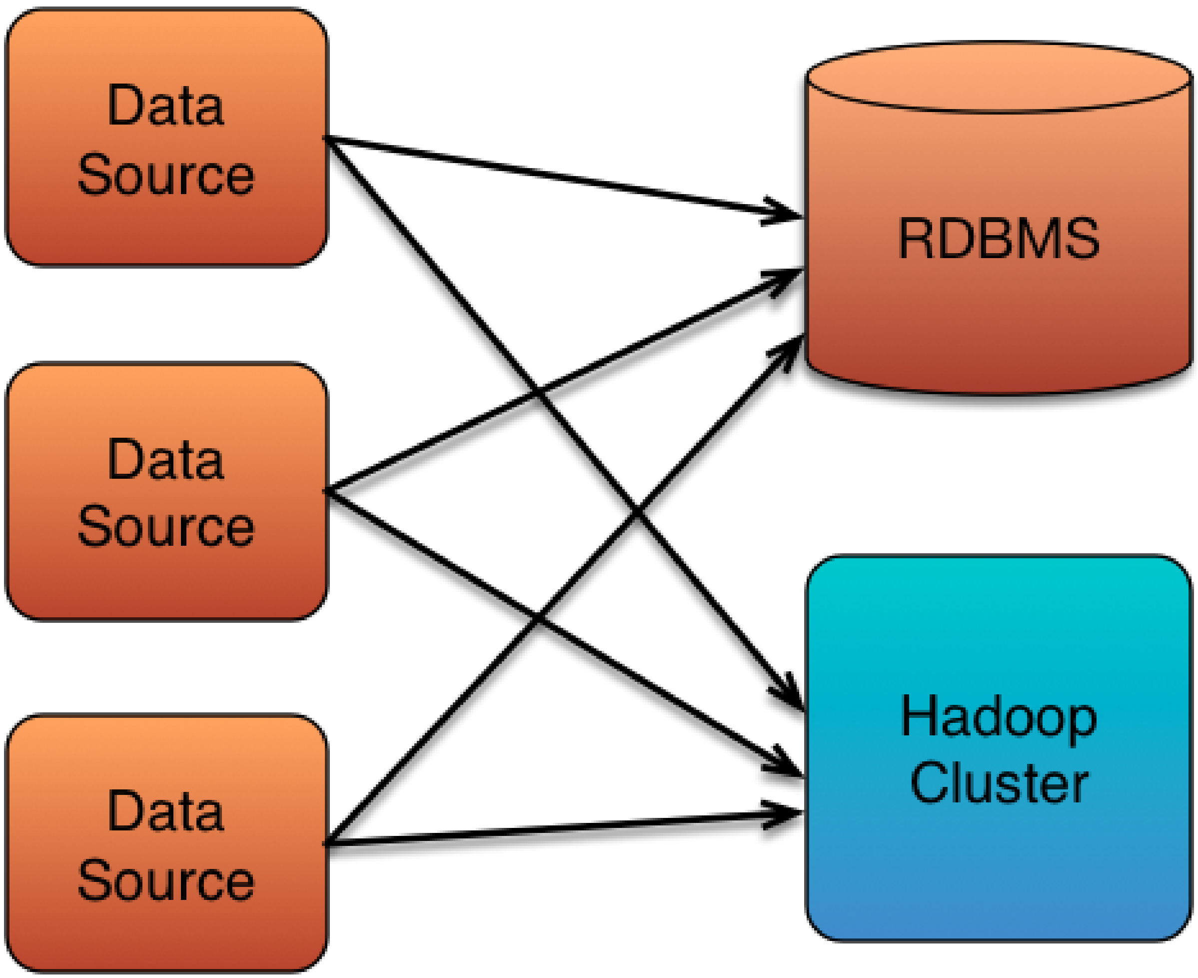
- **Systems Complexity**
- *Real-Time Processing is Becoming Prevalent*
- *Kafka: A Stream Data Platform*
- *Chapter Summary*

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

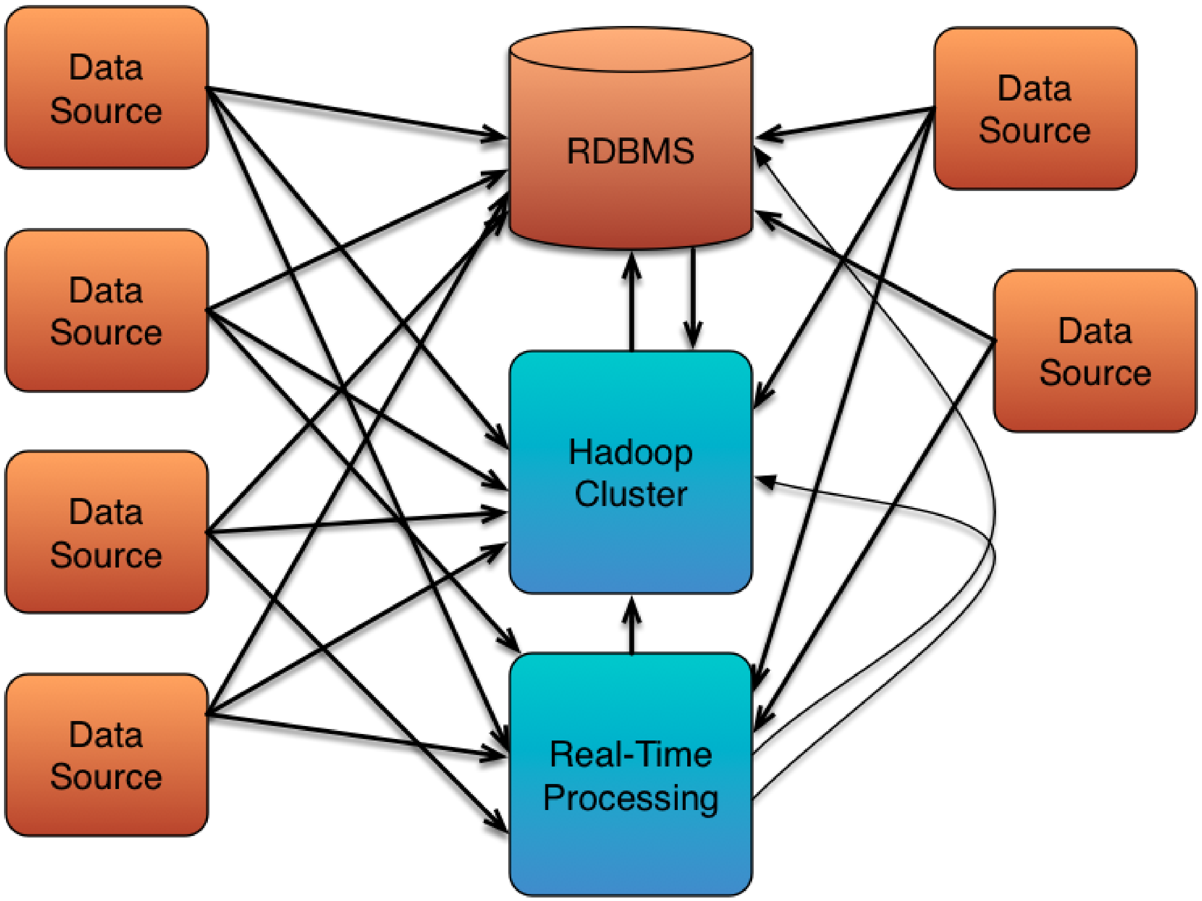
Small Numbers of Systems are Easy to Integrate

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



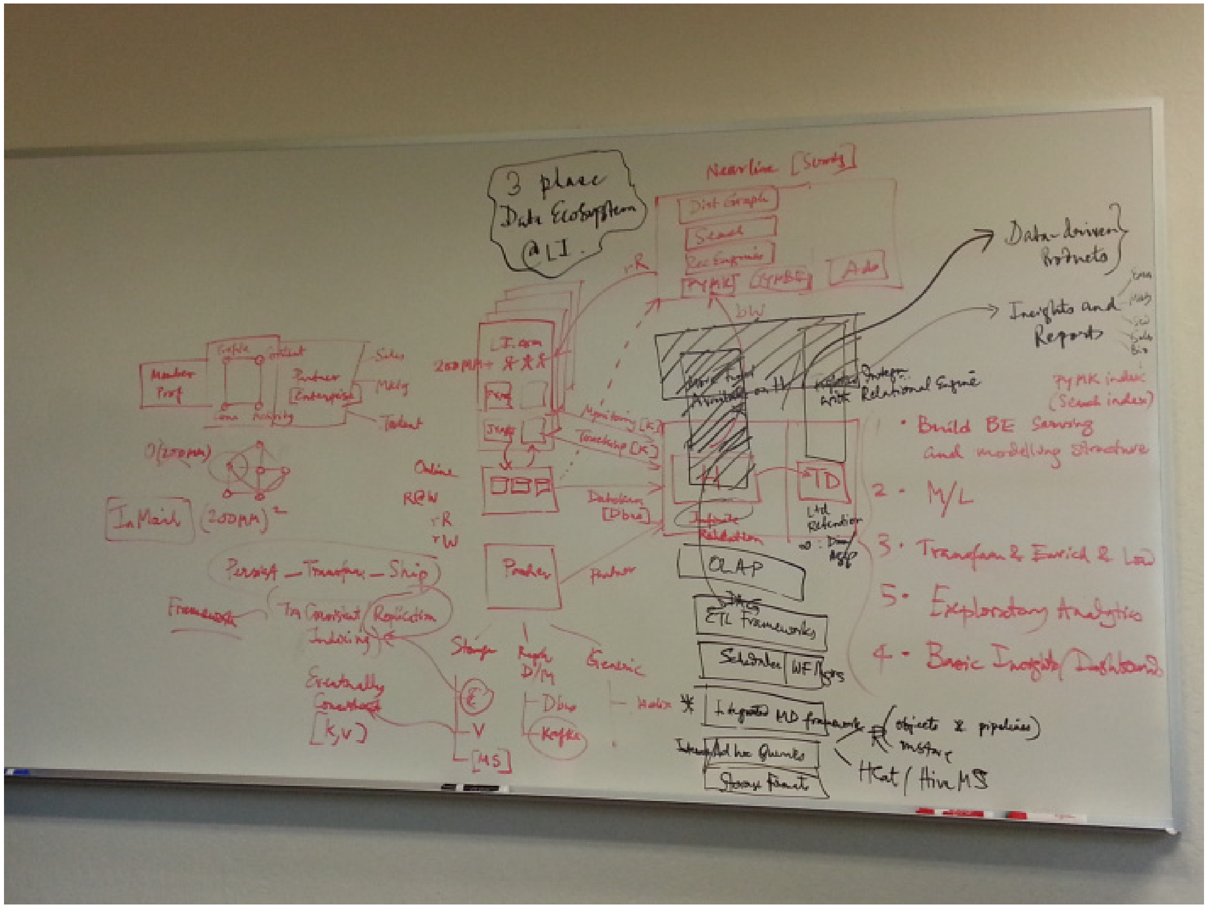
More Systems Rapidly Introduce Much More Complexity (1)

- **As we add more systems, complexity increases dramatically**



More Systems Rapidly Introduce Much More Complexity (2)

- **...until eventually things become unmanageable**



The Motivation for Apache Kafka

- *Systems Complexity*
- **Real-Time Processing is Becoming Prevalent**
- *Kafka: A Stream Data Platform*
- *Chapter Summary*

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 processing allows real-time decisions
- **Examples:**
  - 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

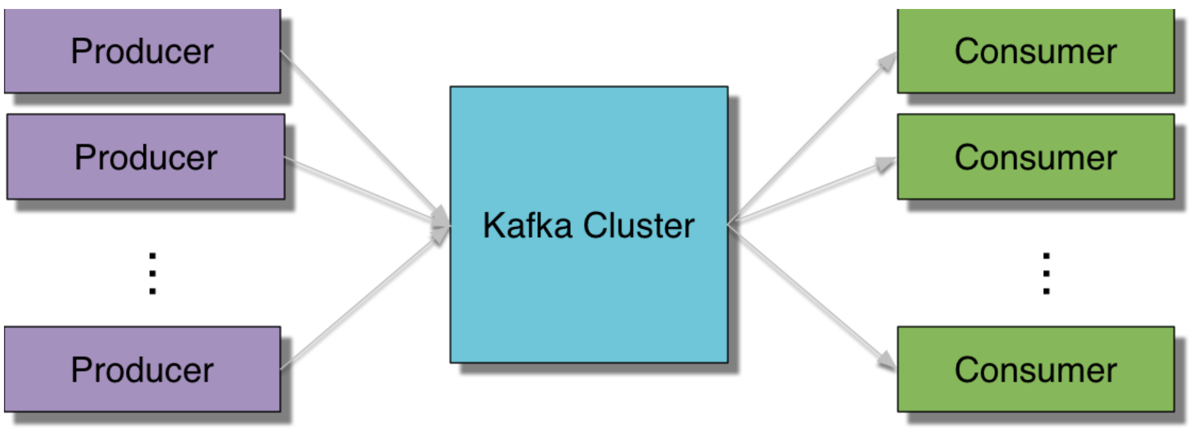
- *Systems Complexity*
- *Real-Time Processing is Becoming Prevalent*
- **Kafka: A Stream Data Platform**
- *Chapter Summary*

Kafka's Origins

- **Kafka was designed to solve both problems**
  - Simplifying data pipelines
  - Handling streaming data
- **Originally created at LinkedIn**
  - Now at the core of LinkedIn's architecture
    - Processes over 1.4 *trillion* messages per day
- **An open source, top-level Apache project since 2012**

A Universal Pipeline for Data

- **Kafka decouples data source and destination systems**
  - Via a *publish/subscribe* architecture
- **Using Kafka, all data sources write their data to the Kafka cluster**
- **Any system wishing to use the data reads it from Kafka**



- **Data sources are known as *Producers***
- **Systems reading the data are *Consumers***

Multiple Consumers for Multiple Use-Cases

- **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.
- **Increasing the number of Consumers does not add significant load to the system**
- **Adding a new Consumer does not require any modification to the Producer(s)**
  - The new Consumer simply needs to know how the data being created by the Producer(s) is formatted

Key Kafka Features (1)

- **Producers write data in the form of *messages* to the Kafka cluster**
  - Messages are key-value pairs, though the Producer does not need to generate the key itself
- **Messages are written to *topics***
  - Topics provide us with a way of splitting messages into logical groups
- **Consumers read messages from one or more topics**
  - Multiple consumers can be combined into a *consumer group*
  - Consumer groups provide scaling capabilities (more on this later)
- **Data retention time in Kafka can be configured on a per-topic basis**

Key Kafka Features (2)

- **Kafka is very scalable, and very resilient**
  - 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
  - Messages are replicated on multiple machines for reliability (more on this later)
- **Consumers can be shut down temporarily**
  - When they restart, they will continue to read from where they left off

The Motivation for Apache Kafka

- *Systems Complexity*
- *Real-Time Processing is Becoming Prevalent*
- *Kafka: A Stream Data Platform*
- **Chapter Summary**

Chapter Summary

- Kafka was designed to simplify data pipelines, and to provide a way for systems to process streaming data
- Producers write data to the Kafka cluster
- Consumers read data from the cluster
- Adding new Consumers does not require Producers to be modified
- Kafka is extremely performant, scalable, and reliable

Kafka Fundamentals

Chapter 03

Course Contents

Kafka Fundamentals

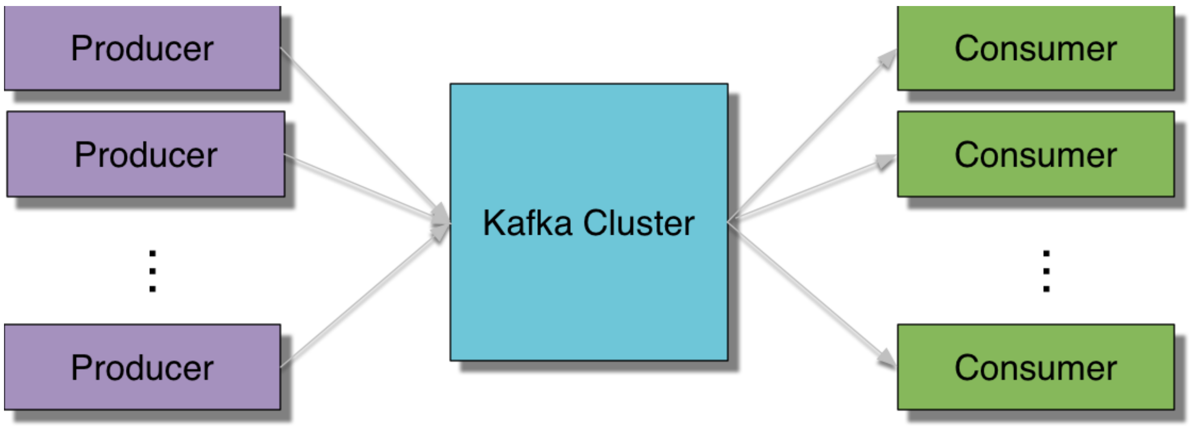
- **In this chapter you will learn:**
  - How Producers write data to a Kafka cluster
  - How data is divided into partitions, and then stored on Brokers
  - How Consumers read data from the cluster
  - What ZooKeeper is, and how it is used by Kafka clusters

Kafka Fundamentals

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

Reprise: A Very High-Level View of Kafka

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



Messages and Topics

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

Kakfa Components

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

Kafka Fundamentals

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

Producer Basics

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

Kafka Messages

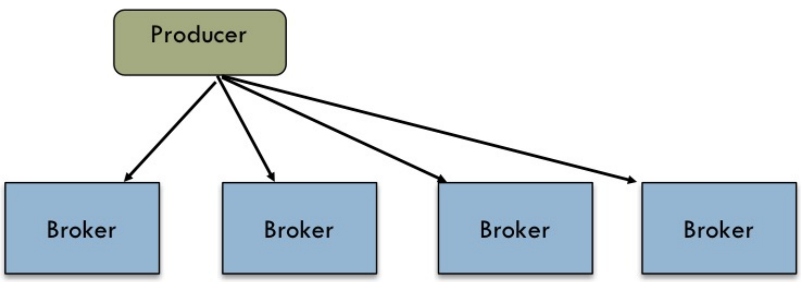
- **A message is the basic unit of data in Kafka**
- **A message is a key-value pair**
- **Key and value can be any data type**
  - You provide a serializer to turn the key and value into byte arrays
- **Key is optional**
  - Keys are used to determine which *Partition* (see later) a message will be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

Topics

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

Topic Partitions Provide Scalability

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



Kafka Fundamentals

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

Broker Basics

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

Brokers Manage Partitions

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

Kafka Fundamentals

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

Consumer Basics

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

The Advantages of a Pull Architecture

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

Keeping Track of Position

- **As messages are written to a topic, the Consumer will automatically retrieve them**
- **The *Consumer Offset* keeps track of the latest message read**
- **If necessary, the Consumer Offset can be changed**
  - For example, to reread messages
- **The Consumer Offset is stored in a special Kafka topic**
  - (Aside: Previously offsets were stored in ZooKeeper, but as of Kafka 9.0 this is no longer the case)

Kafka Fundamentals

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

What is ZooKeeper?

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

How Kafka Uses ZooKeeper

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

Kafka Fundamentals

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

Decoupling Producers and Consumers

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

The Page Cache for High Performance

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

Kafka Fundamentals

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

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

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

Kafka Fundamentals

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

Chapter Summary

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

Kafka’s Architecture

Chapter 04

Course Contents

Kafka's Architecture

- **In this chapter you will learn:**
  - How Kafka's log files are stored on the Kafka Brokers
  - How Kafka uses replicas for reliability
  - What the read path and write path look like
  - How Consumer Groups and Partitions provide scalability

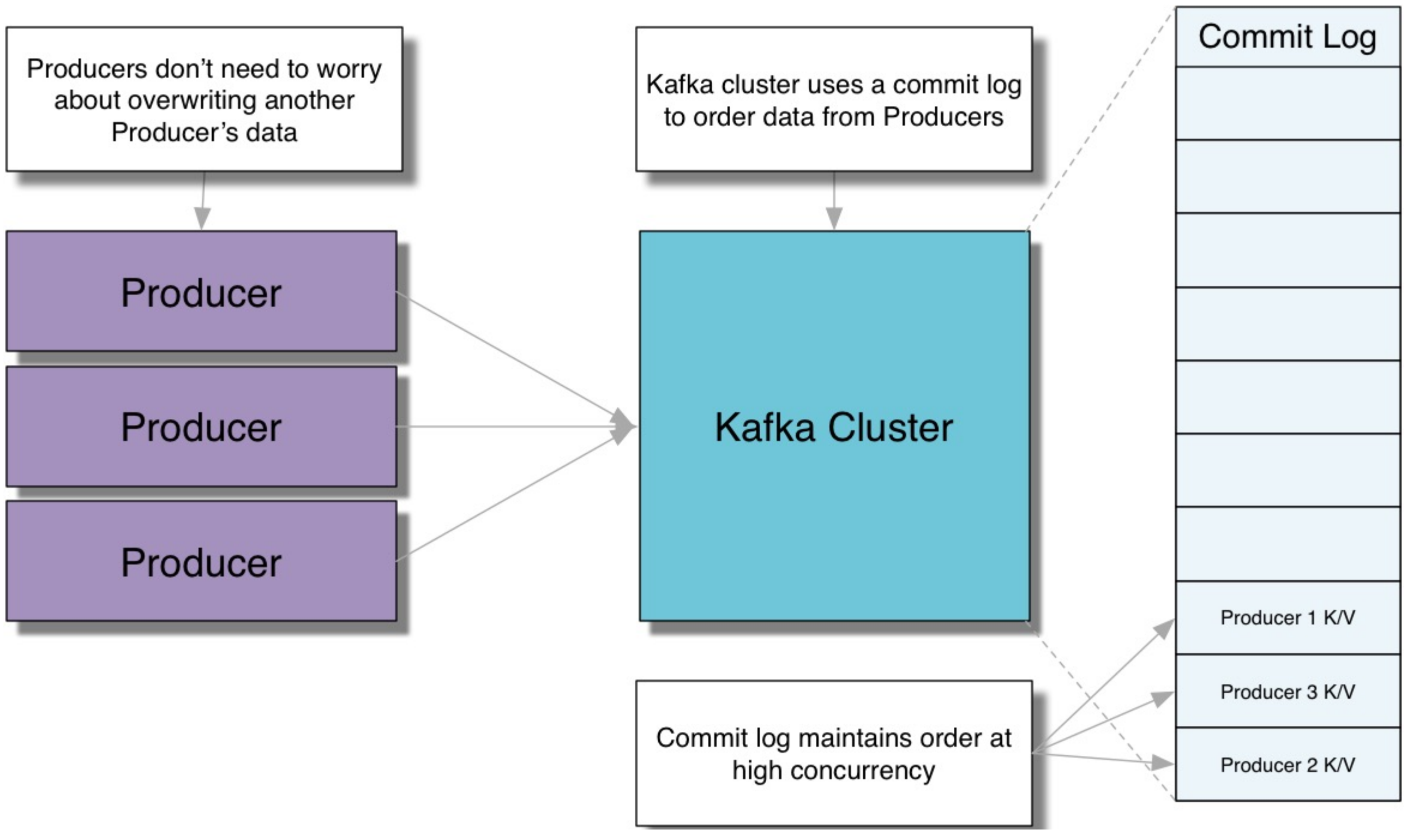
Kafka's Architecture

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

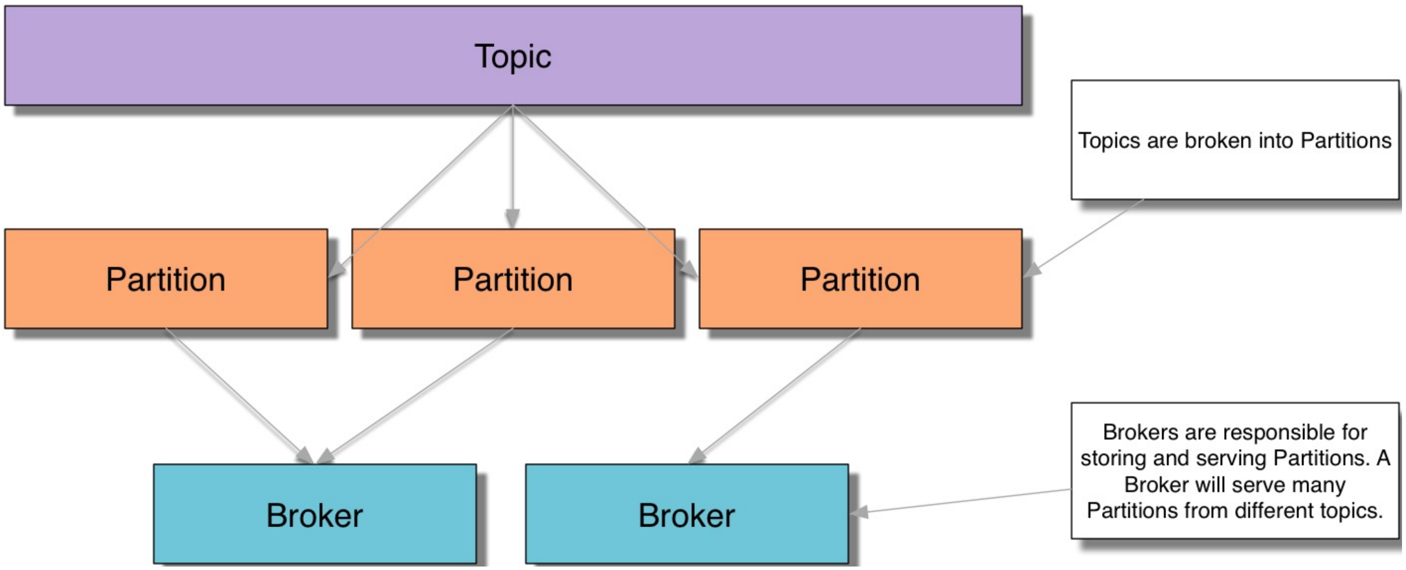
What is a Commit Log?

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

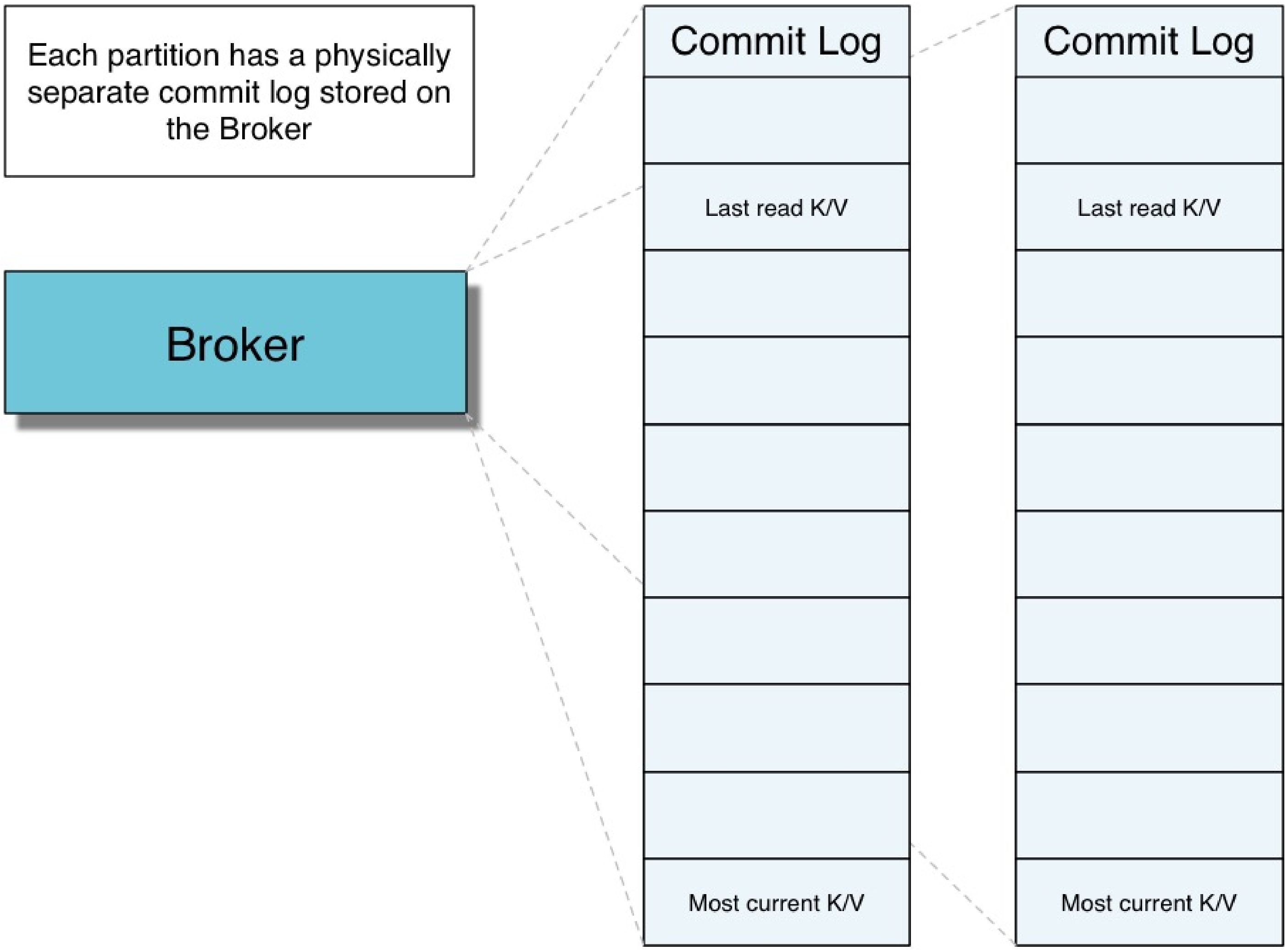
The Commit Log for High Concurrency



Brokers Store Partitions of Topics



Partitions Are Stored as Separate Logs



Kafka's Architecture

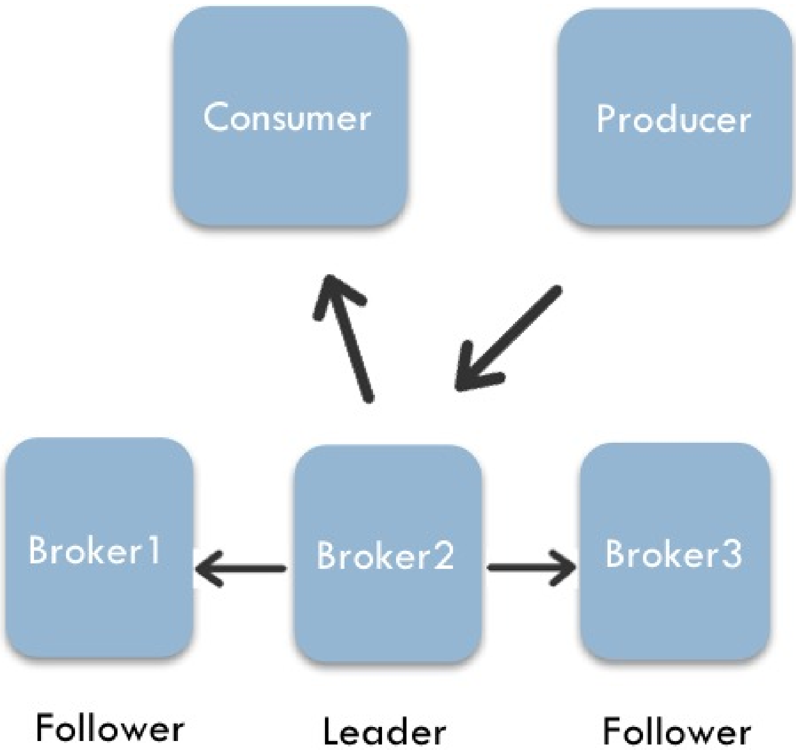
- *Kafka's Log Files*
- **Replicas for Reliability**
- *Kafka's Write Path*
- *Kafka's Read Path*
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Summary*

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


Consumer
Producer
Broker1
Broker2
Broker3
Follower
Leader
Follower

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
- **If a Leader fails, the Kafka cluster will elect a new Leader from among the Followers**
  - Using ZooKeeper

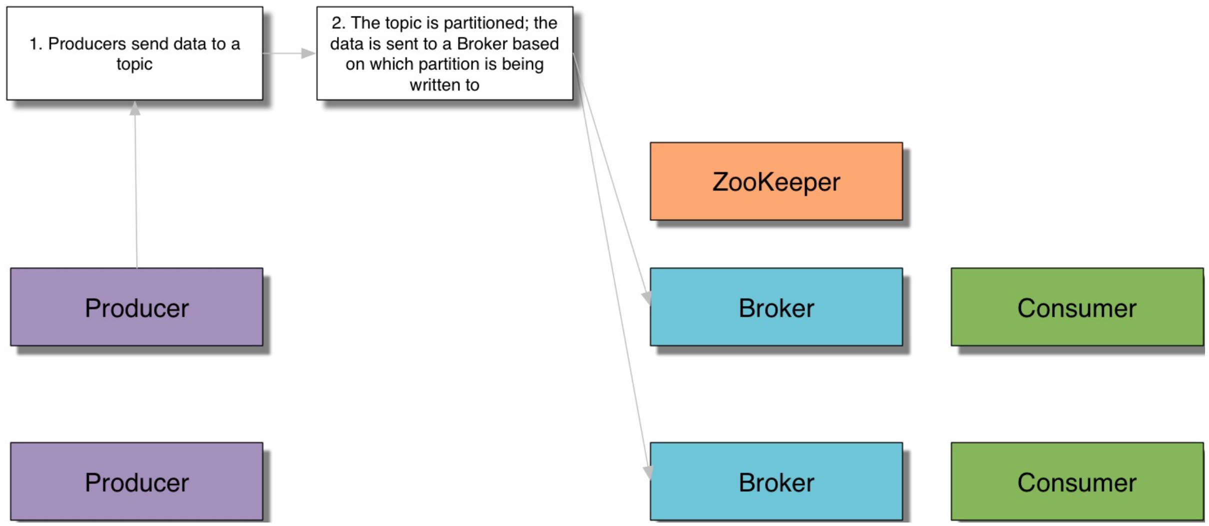
In-Sync Replicas

- **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 IRSs 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**

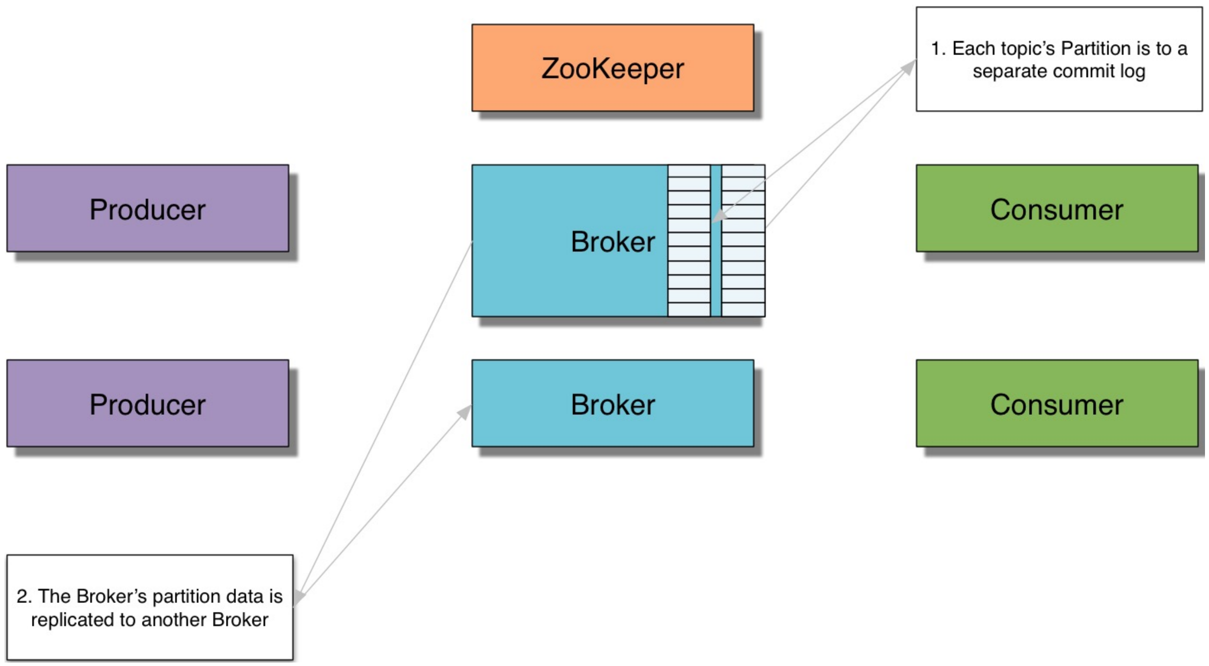
Kafka's Architecture

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

Kafka’s Write Path (1)



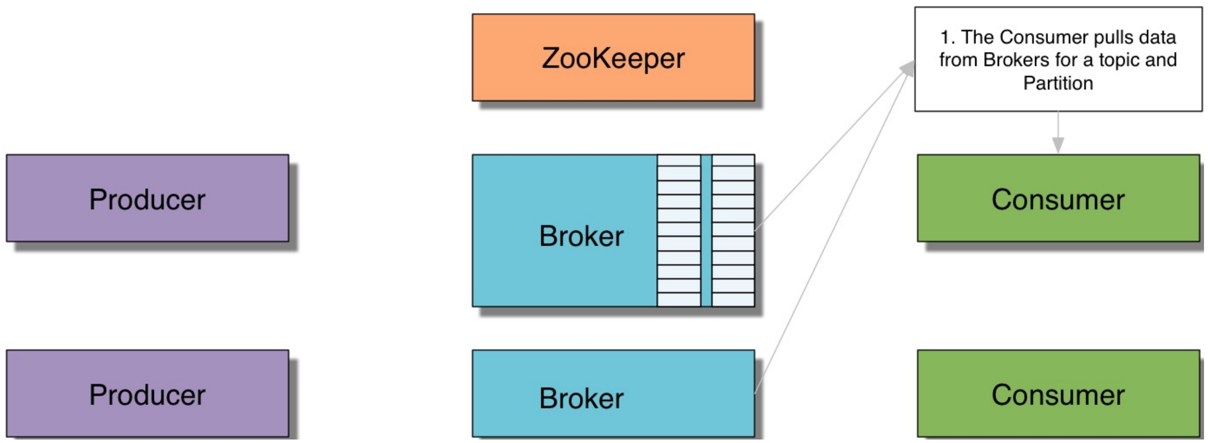
Kafka's Write Path (2)



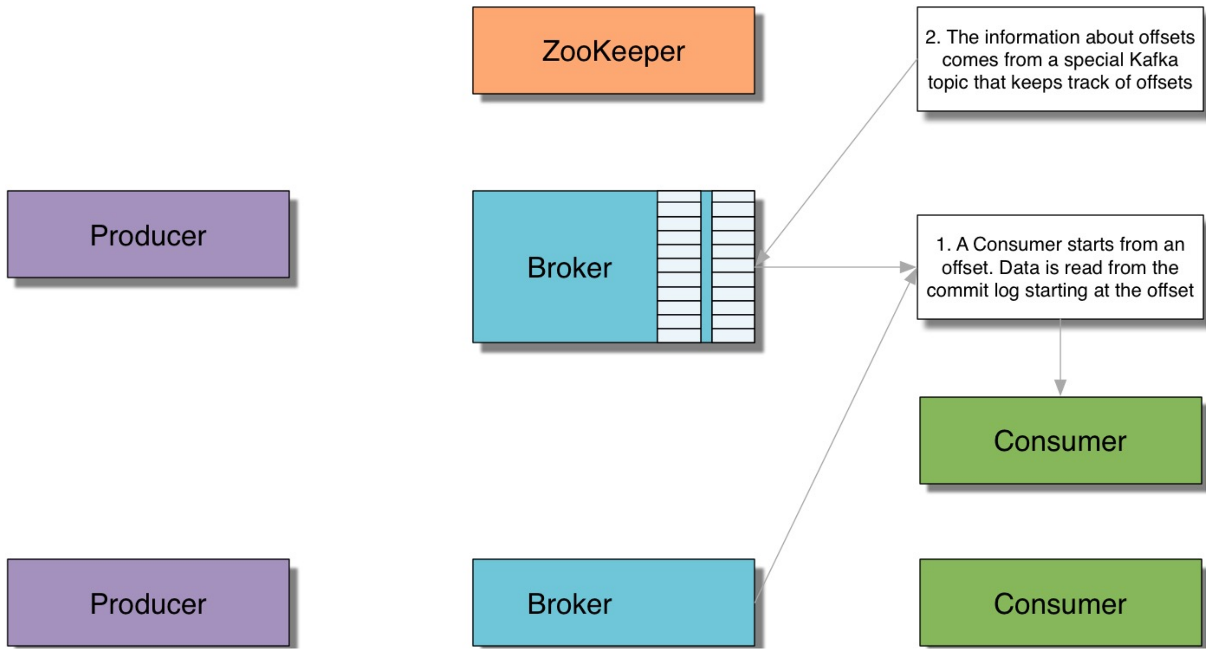
Kafka’s Architecture

- *Kafka’s Log Files*
- *Replicas for Reliability*
- *Kafka’s Write Path*
- **Kafka’s Read Path**
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Summary*

Kafka's Read Path (1)



Kafka's Read Path (2)



Kafka's Architecture

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

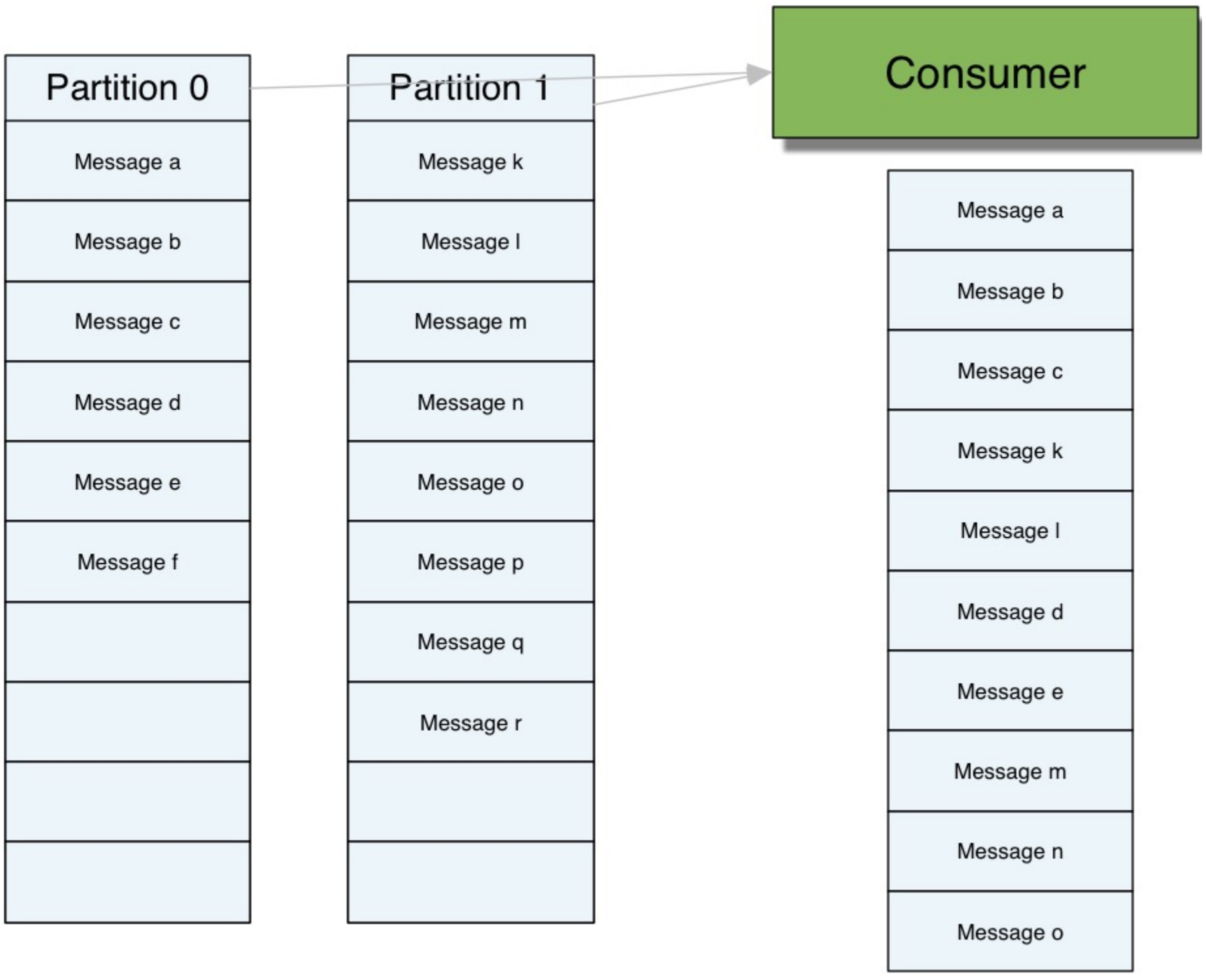
Scaling using Partitions

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

An Important Note About Ordering (1)

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

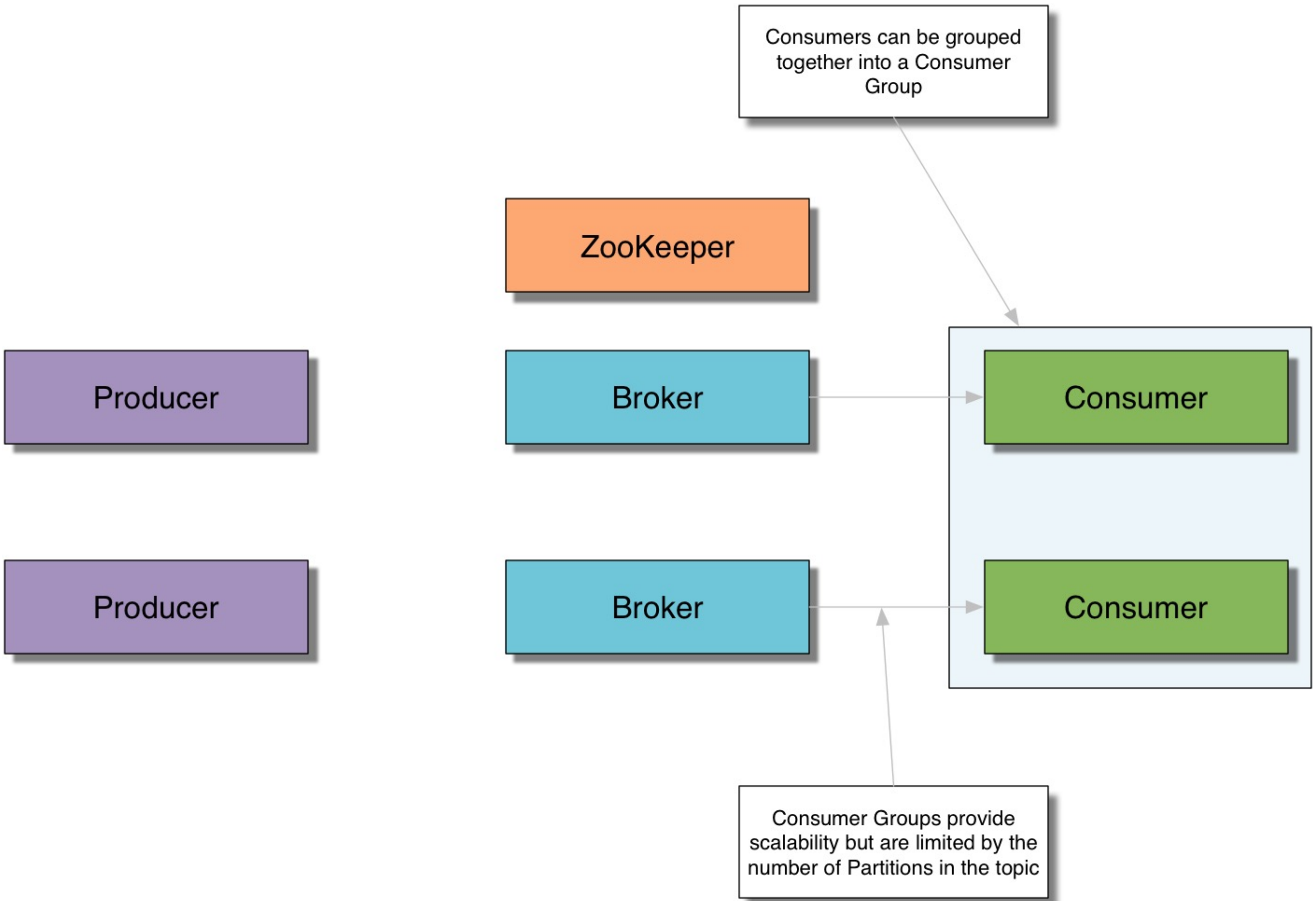
An Important Note About Ordering (2)



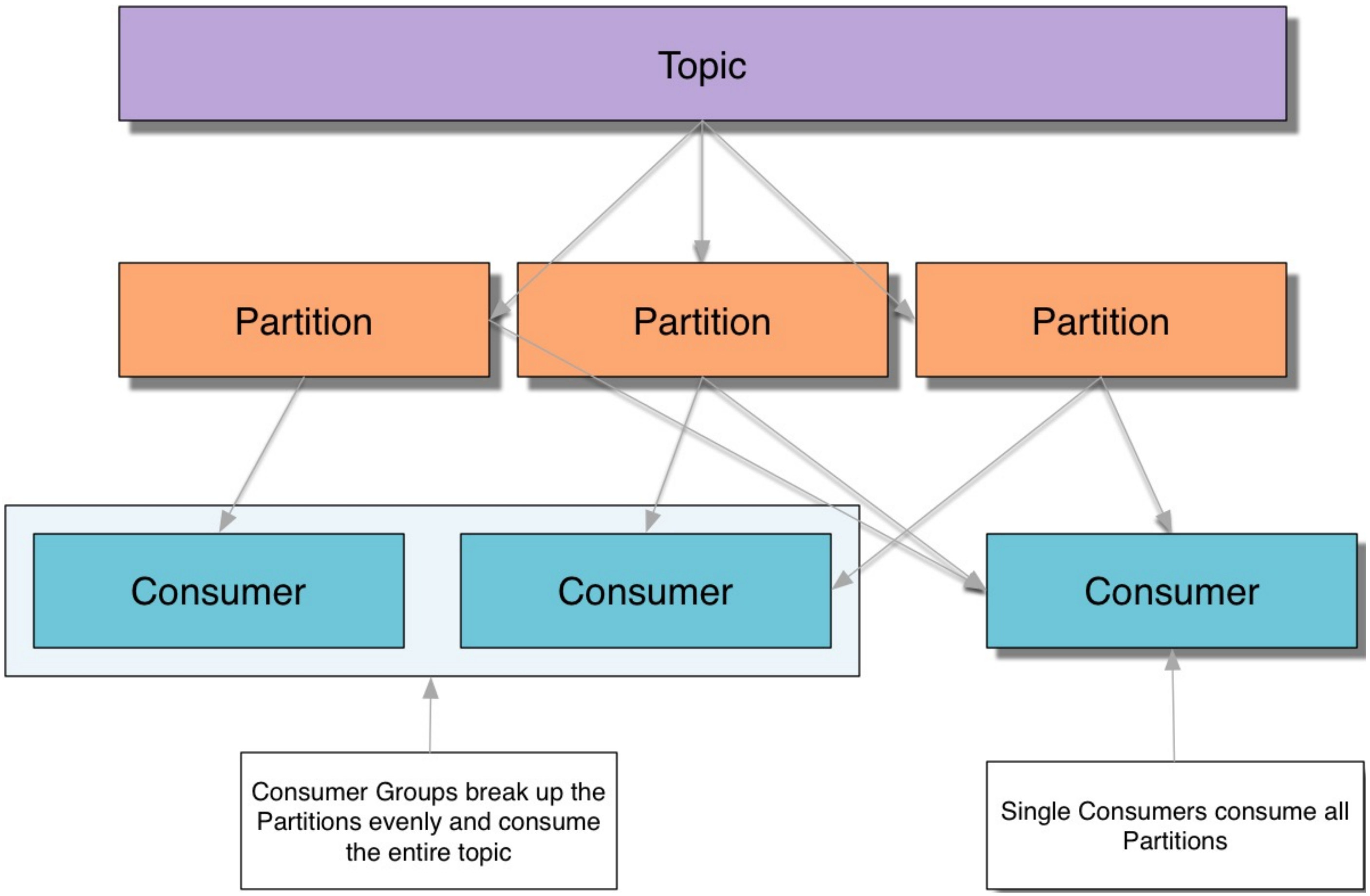
Consumer Groups (1)

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

Consumer Groups (2)



Consumer Groups (3)



Consumer Groups: Limitations

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

Consumer Groups: Caution When Changing Partitions

- **Recall: All messages with the same key will go to the same Consumer**
  - However, if you change the number of Partitions in the topic, this may not be the case
    - Example: Using Kafka's default Partitioner, Messages with key *K1* were previously written to Partition 2 of a topic
    - After repartitioning, new messages with key *K1* may now go to a different Partition
    - Therefore, the Consumer which was reading from Partition 2 may not get those new messages, as they may be read by a new Consumer

Kafka's Architecture

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

Hands-On Exercise: Consuming from Multiple Partitions

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

Kafka's Architecture

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

Chapter Summary

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

Developing With Kafka

Chapter 05

Course Contents

Developing With Kafka

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

Developing With Kafka

- **Using Maven for Project Management**
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *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 Summary*

About Maven

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

Why Use Maven?

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

Sample Maven POM for Kafka (1)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>io.confluent.kafkaprogram</groupId>
  <artifactId>kafkaprogram</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
      <!-- Keep Confluent versions as properties to allow easy modification -->
    <confluent.version>2.0.0</confluent.version>
    <kafka.version>0.9.0.0-cp1</kafka.version>
    <!-- Maven properties for compilation -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
```

Sample Maven POM for Kafka (2)

```
<dependencies>
    <!-- Add the Kafka dependency -->
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>${confluent.version}</version>
  </dependency>
  <dependency>
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>${kafka.version}</version>
  </dependency>
</dependencies>
```

Sample Maven POM for Kafka (3)

```
  <build>
    <plugins>
      <plugin>
          <!-- Set the Java target version to 1.7 -->
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.0</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-eclipse-plugin</artifactId>
        <version>2.9</version>
        <configuration>
            <downloadJavadocs>true</downloadJavadocs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Using Maven (1)

- **To compile a Maven project:**

```
mvn compile
```

- **To run unit tests:**

```
mvn test
```

- **To create a JAR artifact**

```
mvn package
```

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

Using Maven (2)

- **To build and run a project managed by Maven:**

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

- **If your application requires arguments, use:**

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

Creating Eclipse Projects with Maven

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

```
mvn -Declipse.workspace=/home/training/workspace eclipse:configure-workspace
```

- **Create the Eclipse project files for the Maven project**

```
mvn eclipse:eclipse
```

- **You can then import the project into Eclipse**

Maven And Our Exercise Environment

- **One common complaint with Maven is that it “downloads the Internet”**
  - When first run, it will download all dependencies, and *their* dependencies, and so on
- **This can require a significant amount of time, and network bandwidth**
- **To avoid this during class, we use Apache Archiva, which caches files locally**

Developing With Kafka

- *Using Maven for Project Management*
- **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 Summary*

The Kafka API

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

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

- *Using Maven for Project Management*
- *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 Summary*

The Producer API

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

Important `Properties` Elements (1)

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

Important `Properties` Elements (2)

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

Creating the Properties and KafkaProducer Objects

```
Properties props = new Properties;
props.put("bootstrap.servers", "localhost:9090");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<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`

Sending Messages to Kafka

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

producer.close();
```

1 Alternatively:

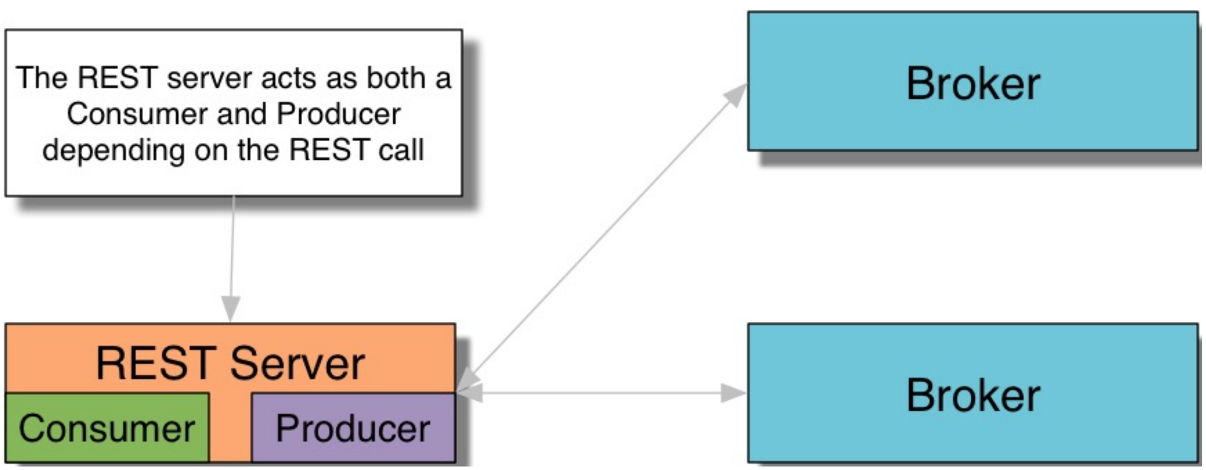
```
producer.send(new ProducerRecord<String, String>("mytopic", k, v));
```

Developing With Kafka

- *Using Maven for Project Management*
- *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 Summary*

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**
  - Base64-encoded JSON for the key and value
- **Uses GET to retrieve data from Kafka**



A Python Producer Using the REST Proxy

```
#!/usr/bin/python

import requests
import base64
import json

url = "http://restserver:8082/topics/my_topic"
headers = {
  "Content-Type" : "application/vnd.kafka.binary.v1+json"
    }
# Create one or more messages
payload = {"records":
  [{
    "key":base64.b64encode("firstkey"),
    "value":base64.b64encode("firstvalue")
  }]}
# Send the message
r = requests.post(url, data=json.dumps(payload), headers=headers)
if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
```

Developing With Kafka

- *Using Maven for Project Management*
- *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 Summary*

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

- *Using Maven for Project Management*
- *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 Summary*

The New Consumer API

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

Consumers and Offsets

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

Important Consumer Properties

- **Important Consumer properties include:**

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

Creating the Properties and KafkaConsumer Objects

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9090");
props.put("group.id", "samplegroup");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my_topic_1", "my_topic_2")); (1)
```

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

Reading Messages from Kafka with `poll()`

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

① **Loop forever**

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

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**

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

Important: The `KafkaConsumer` Is Not Thread-Safe

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

Developing With Kafka

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *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 Summary*

A Python Consumer Using the REST API (1)

```
#!/usr/bin/python

import requests
import base64
import json
import sys

# Base URL for interacting with REST server
baseurl = "http://restserver:8082/consumers/group1" ①

# Create the Consumer instance
print "Creating consumer instance"
payload = {
  "format": "binary"
  }
headers = {
"Content-Type" : "application/vnd.kafka.v1+json"
  }
```

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

A Python Consumer Using the REST API (2)

```python
r = requests.post(baseurl, data=json.dumps(payload), headers=headers)

if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
  sys.exit("Error thrown while creating consumer")

# Base URI is used to identify the consumer instance
base_uri = r.json()["base_uri"]
```

A Python Consumer Using the REST API (3)

```
# Get the message(s) from the Consumer
headers = {
  "Accept" : "application/vnd.kafka.binary.v1+json"
    }

# Request messages for the instance on the topic
r = requests.get(base_uri + "/topics/my_topic", headers=headers, timeout=20)

if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
  sys.exit("Error thrown while getting message")
```

A Python Consumer Using the REST API (4)

```
# Output all messages
for message in r.json():
  if message["key"] is not None:
    print "Message Key:" + base64.b64decode(message["key"])
  print "Message Value:" + base64.b64decode(message["value"])

# When we're done, delete the Consumer
headers = {
  "Accept" : "application/vnd.kafka.v1+json"
    }

r = requests.delete(base_uri, headers=headers)

if r.status_code != 204:
  print "Status Code: " + str(r.status_code)
  print r.text
```

Developing With Kafka

- *Using Maven for Project Management*
- *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 Summary*

Hands-On Exercise: Writing a Basic Consumer

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

Developing With Kafka

- *Using Maven for Project Management*
- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *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 Summary**

Chapter Summary

- **The Kafka API provides Java clients for Producers and Consumers**
- **Client libraries for other languages are available, though the quality varies**
  - Confluent supports a C/C++ client, and others are being developed
- **Confluent's REST Proxy allows other languages to access Kafka without the need for native client libraries**

More Advanced Kafka Development

Chapter 06

Course Contents

More Advanced Kafka Development

- **In this chapter you will learn:**
  - How to write a multi-threaded Consumer in Java
  - How to specify the offset to read from
  - How to manually commit reads from the Consumer
  - How to create a custom Partitioner
  - How to control message delivery reliability

More Advanced Kafka Development

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

Limitations of our Previous Consumer

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

Creating a Multi-Threaded Consumer (1)

- **We use the Runnable Interface**

```
public class MyConsumerLoop implements Runnable {
  private KafkaConsumer<String, String> consumer;
  private List<String> topics;
  private int id;

  public MyConsumerLoop(int id, String groupId, List<String> topics) {
    this.id = id;
    this.topics = topics;
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9090");
    props.put("group.id", groupId);
    props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
    this.consumer = new KafkaConsumer<>(props);
  }
```

Creating a Multi-Threaded Consumer (2)

```
public void run() {
    try {
      consumer.subscribe(topics);

      while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE); ①
        for (ConsumerRecord<String, String> record : records) {
            // Do something with the message!
        }
      }
    } catch (WakeupException e) {
      // ignore for shutdown
    } finally {
      consumer.close();
    }
  }

  public void shutdown() {
    consumer.wakeup();
  }
}
```

① **This will cause `poll` to block indefinitely**

Creating a Multi-Threaded Consumer (3)

- **Now instantiate five threads and start them**

```
public static void main(String[] args) {
  int numConsumers = 5;
  String groupId = "my-consumer-group"
  List<String> topics = Arrays.asList("mytopic");
  ExecutorService executor = Executors.newFixedThreadPool(numConsumers);

  List<MyConsumerLoop> consumers = new ArrayList<>();
  for (int i = 0; i < numConsumers; i++) {
    MyConsumerLoop consumer = new MyConsumerLoop(i, groupId, topics);
    consumers.add(consumer);
    executor.submit(consumer);
  }
```

Creating a Multi-Threaded Consumer (3)

- **Finally, configure what to do when the process terminates**

```
Runtime.getRuntime().addShutdownHook(new Thread() {
  @Override
  public void run() {
    for (MyConsumerLoop consumer : consumers) {
      consumer.shutdown();
    }
    executor.shutdown();
    try {
      executor.awaitTermination(5000, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
      e.printStackTrace;
    }
  }
});
}
```

More Advanced Kafka Development

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

Determining the Offset When a Consumer Starts

- **The Consumer property `auto.offset.reset` determines what to do if there is no offset in Kafka for the Consumer's Consumer Group**
  - For example, the first time a particular Consumer Group starts
- **The value can be one of:**
  - `earliest`: Automatically reset the offset to the earliest available
  - `latest`: Automatically reset to the latest offset available
  - non: Throw an exception if no previous offset can be found for the ConsumerGroup
- **The default is `latest`**

Changing the Offset Within the Consumer (1)

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

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:**

```
...
consumer.subscribe("mytopic");
consumer.poll(0);
for (TopicPartition partition: consumer.assignment()) { ①
    consumer.seek(partition, 0);
}
```

① `assignment()` returns a list of the partitions currently assigned to this Consumer

More Advanced Kafka Development

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

Adding Consumers Will Cause Rebalancing

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

The Case For and Against Rebalancing

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

More Advanced Kafka Development

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

Default Consumer Behavior: Automatically Committed Offsets

- **By default, `enable.auto.commit` is set to `true`**
  - 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**

Example: 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**

Manually Committing Offsets

- **You can manually commit offsets by calling `consumer.commitSync()`**
  - Commits the offsets returned on the last `poll()` for all subscribed topics and partitions
  - Ensure that you process all the records returned by `poll()`, or you may miss messages
- **`commitSync()` blocks until it succeeds**
  - It retries as long as it does not receive a fatal error
- **`commitAsync()` is also available**
  - Returns immediately
  - Optionally takes a callback that will be triggered when the Broker responds

Manually Committing Offsets From the REST Proxy

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

```
payload = {
  "format": "binary",
  # Manually/Programmatically commit offset
  "enable.auto.commit": "false"
  }

headers = {
"Content-Type" : "application/vnd.kafka.v1+json"
  }

r = requests.post(baseurl, data=json.dumps(payload), headers=headers)
```

```
# Commit the offsets
if shouldCommit() == True:
  r = requests.post(base_uri + "/offsets", headers=headers, timeout=20)
  if r.status_code != 200:
    print "Status Code: " + str(r.status_code)
    print r.text
    sys.exit("Error thrown while committing")
  print "Committed"
```

Storing Offsets Outside of Kafka

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

More Advanced Kafka Development

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

Kafka's Default Partitioning Scheme

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

Creating a Custom Partitioner

- **To create a custom Partitioner, you shoudl 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, serialzed 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**

```
public class MyPartitioner implements Partitioner {
    public void configure(Map<String, ?> configs) {}
    public void close() {}

    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes, Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceOf String)))
            throw new InvalidRecordException("Record did not have a string Key");

        if (((String) key).equals("OurBigKey")) ①
            return 0; // This key will always go to Partition 0

        // Other records will go to the rest of the partitions using a hashing function
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;
    }
}
```

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

An Alternative to a Custom Partitioner

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

More Advanced Kafka Development

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

Replication Factor Affects Message Durability

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

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**

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

More Advanced Kafka Development

- *Creating a Multi-Threaded Consumer*

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

Hands-On Exercise: Accessing Previous Data

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

More Advanced Kafka Development

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

Chapter Summary

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

Schema Management In Kafka

Chapter 07

Course Contents

Schema Management In Kafka

- **In this chapter you will learn:**
  - What Avro is, and how it can be used for data with a changing schema
  - How to write Avro messages to Kafka
  - How to use the Schema Registry for better performance

Schema Management In Kafka

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

What is Seralization?

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

The Need for a More Complex Serialization System

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

Avro: An Efficient Data Serialization System

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

Schema Management In Kafka

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

Avro Schemas

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

Avro Data Types (Simple)

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

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

Avro Data Types (Complex)

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

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

Example Avro Schema (1)

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

Example Avro Schema (2)

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

Example Schema Snippet with `array` and `map`

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

Example Schema Snippet with enum

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

Schema Evolution

- **Often, Avro schemas evolve as updates to code happen**
- **We often want compatibility between schemas:**
- **Backward compatibility**
  - Code with a new version of the schema can read data written in the old schema
- **Forward compatibility**
  - Code with previous versions of the schema can read data written using the new schema

Integrating Avro Into Your Maven Project (1)

- **To use Avro with your Maven project:**

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

Integrating Avro Into Your Maven Project (2)

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

Schema Management In Kafka

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

What Is the Schema Registry?

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

Java Avro Producer example

```java
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092");
// Configure serializer classes
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
          io.confluent.kafka.serializers.KafkaAvroSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
          io.confluent.kafka.serializers.KafkaAvroSerializer.class);
// Configure schema repository server
props.put("schema.registry.url", "http://schemaregistry1:8081");
// Create the producer expecting Avro objects
KafkaProducer<Object, Object> avroProducer = new KafkaProducer<Object, Object>(props);
// Create the Avro objects for the key and value
CardSuit suit = new CardSuit("spades");
SimpleCard card = new SimpleCard("spades", "ace");
// Create the ProducerRecord with the Avro objects and send them
ProducerRecord<Object, Object> record = new
ProducerRecord<Object, Object>(
    "my_avro_topic", suit, card);
avroProducer.send(record);
```

Java Avro Consumer Example

```java
public class CardConsumer {
  public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", "testgroup");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("session.timeout.ms", "30000");
    props.put("key.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
    props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
    props.put("schema.registry.url", "http://schemaregistry1:8081");
    props.put("specific.avro.reader", "true");

    KafkaConsumer<CardSuit, SimpleCard> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Arrays.asList("my_avro_topic"));

    while (true) {
      ConsumerRecords<CardSuit, SimpleCard> records = consumer.poll(100);
        for (ConsumerRecord<CardSuit, SimpleCard> record : records) {
          System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key().getSuit(), record.value().getCard());

        }
    }
  }
}
```

Python Avro Producer Example

```python
# Read in the Avro files
key_schema = open("my_key.avsc", 'rU').read()
value_schema = open("my_value.avsc", 'rU').read()
```

```python
producerurl = "http://kafkarest1:8082/topics/my_avro_topic"
headers = {
  "Content-Type" : "application/vnd.kafka.avro.v1+json"
}
payload = {
  "key_schema": key_schema,
  "value_schema": value_schema,
  "records":
  [{
    "key": {"suit": "spades"},
    "value": {"suit": "spades", "card": "ace"}
  }]}
# Send the message
r = requests.post(producerurl, data=json.dumps(payload), headers=headers)
if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
```

Python Avro Consumer Example

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

Command-line Consumer Example

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

Schema Management In Kafka

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

Hands-On Exercise: Using Kafka with Avro

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

Schema Management In Kafka

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

Chapter Summary

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

Kafka Connect for Data Movement

Chapter 08

Course Contents

Kafka Connect for Data Movement

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

Kafka Connect for Data Movement

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

What is Kafka Connect?

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

Why Not Just Use Producers and Consumers?

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

Sample Use-Cases

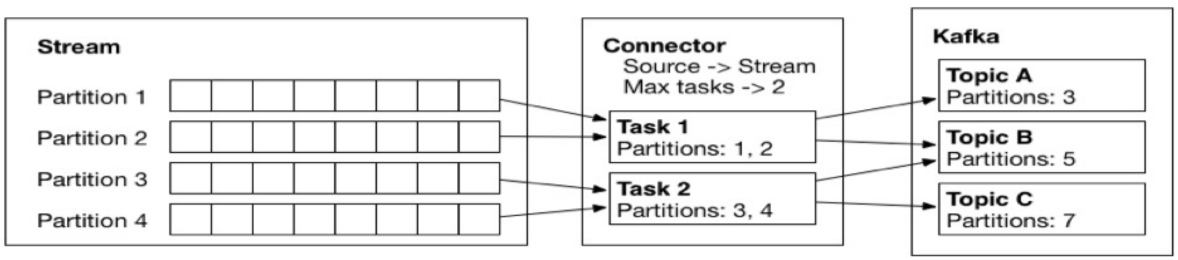
- **Sample use-cases for Kafka Connect:**
  - Stream an entire SQL database, or just specific tables, into Kafka
  - Stream data from Kafka topics into HDFS (the Hadoop Distributed File System) for batch processing
  - Stream data from Kafka topics into ElasticSearch for secondary indexing
  - Stream data from Kafka topics into Cassandra
  - ...

Kafka Connect for Data Movement

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

Kafka Connect Terminology (1)

- **Kafka Connect has three main components:**
  - Connectors
    - Sources and Sinks
  - Tasks
  - Workers



Kafka Connect Terminology (2)

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

Kafka Connect Terminology (3)

- **Source Connectors**

- **Sink Connectors**

Kafka Connect for Data Movement

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

Standalone and Distributed Modes

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

Running in Standalone Mode

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

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

- **Each Connector instance will run in its own thread**

Kafka Connect for Data Movement

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

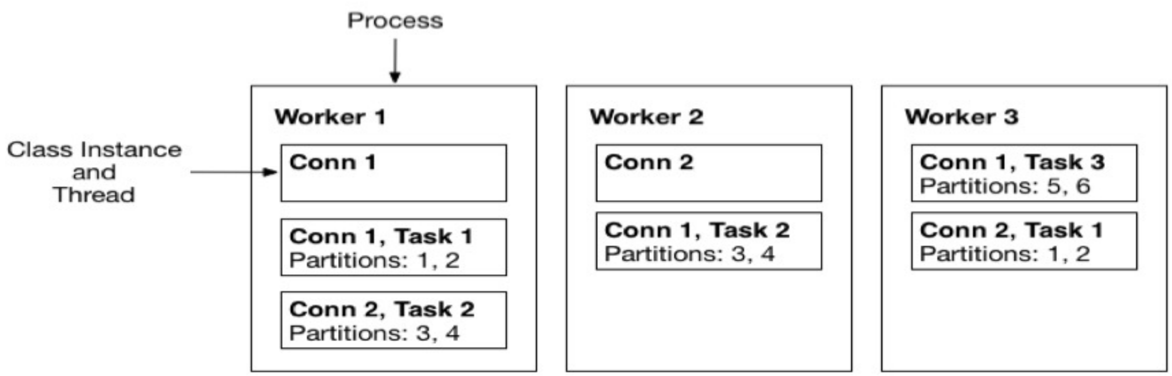
Hands-On Exercise: Running Kafka Connect in Standalone Mode

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

Kafka Connect for Data Movement

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

Workers in Distributed Mode



Running in Distributed Mode

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

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

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

Managing Distributed Mode

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

The REST API

- **A subset of the REST API is shown below**

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

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

Kafka Connect for Data Movement

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

Hands-On Exercise: Using the Kafka Connect REST API

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

Kafka Connect for Data Movement

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

Tracking Source and Sink Offsets (1)

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

Tracking Source and Sink Offsets (2)

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

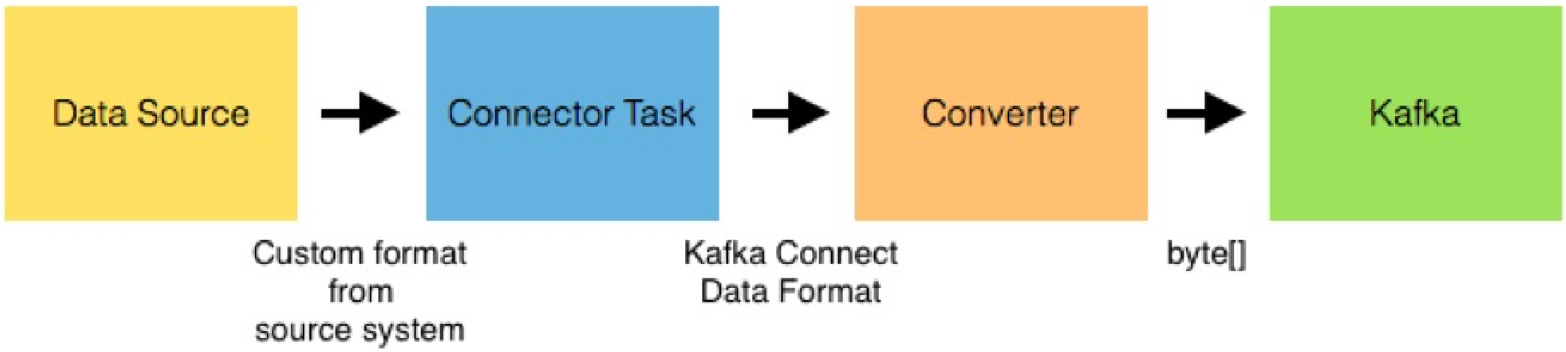
Kafka Connect for Data Movement

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

Converters

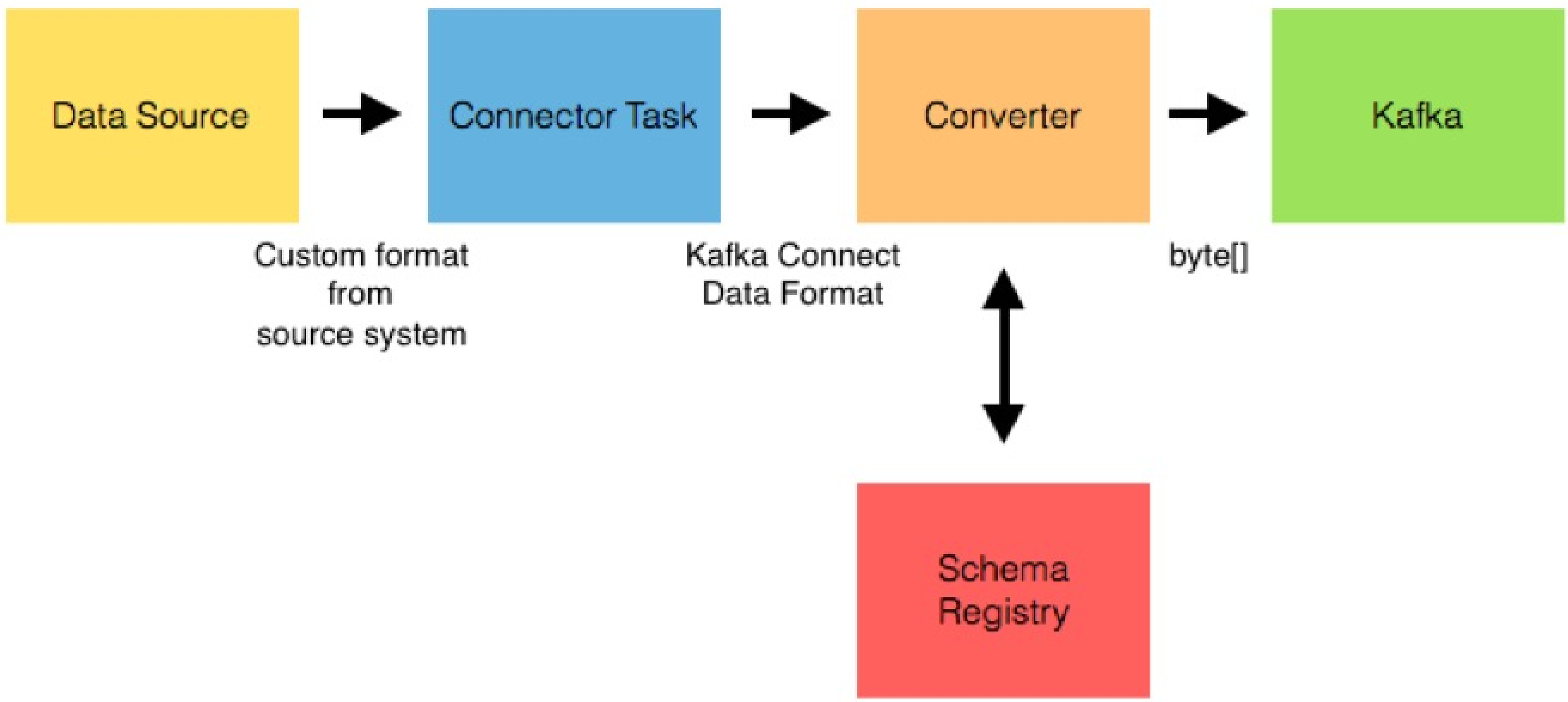
- **Kafka Connect provides a pluggable Converter API**
  - Converts data as it is transferred between the data system and Kafka
- **Source Converters**
  - Invoked between being fetched from the source and then Produced into Kafka
- **Sink Converters**
  - Invoked between Consuming the data from Kafka and writing it to the Sink
- **Converters apply to both the key and value of the message**

The Source Converter Model



The Source Converter with the Schema Registry

- **If an Avro converter is used, the Schema Registry can be used to manage Kafka message schemas**



Kafka Connect for Data Movement

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

Connector Configuration (1)

- **Features to consider when configuring Connectors:**
  - Distributed mode vs Standalone mode
  - Connectors
  - Workers
  - Overriding Producer and Consumer settings

Connector Configuration (2)

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

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

Configuring Workers

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

Configuring Workers: Both Modes

- **Important configuration options common to all workers:**

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

Configuring Workers: Standalone Mode

Parameter	Description
`offset.storage.file.filename`	The file in which to store the Connector offsets

Configuring Workers: Distributed Mode

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

Off-The-Shelf Connectors

- **Connectors included in Confluent Platform:**
  - Local file Source and Sink
  - JDBC Source
  - HDFS Sink

Local File Source and Sink Connector

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

JDBC Source Connector: Overview

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

JDBC Source Connector: Detecting New and Updated Rows

- **The Connector can detect new and updated rows in 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
Bulk	Unfiltered, and not incremental. Used only for one-time bulk loads

JDBC Source Connector: Configuration

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

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

HDFS Sink Connector

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

HDFS Sink Connector: Overview (2)

- **Data format is extensible**
  - Avro, Parquet, custom formats
- **Supports schema evolution and different schema compatibility rules**
- **Works with secure HDFS and the Hive Metastore, using Kerberos**
- **Pluggable Partitioner, supporting:**
  - Kafka Partitioner (default)
  - Field Partitioner
  - Time Partitioner
  - Custom Partitioners

HDFS Sink Connector: Schema Evolution

- **Supports schema evolution and reacts to schema changes in a configurable way**
- **Schema change options:**

Mode	Description
`NONE`	On schema change, commits current set of files for affected topics and writes data with the new schema in new files. (Default)
`BACKWARD`	Latest schema is used to query all data uniformly. Old data is projected to the latest schema
`FORWARD`	Oldest schema is used to query all data uniformly. Data is projected to the oldest schema before writing to HDFS
`FULL`	Old data is read with the new schema. New data is read with the old schema. The Connector performs the same action as **`BACKWARD`**

Kafka Connect for Data Movement

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

Kafka Connect vs Alternative Options (1)

- **There are three categories of systems similar to Connect:**
  - Log and metric collection, processing, and aggregation
    - *e.g.*, Flume, Logstash, Fluentd, Heka
  - ETL tools
    - *e.g.*, Morphlines, HIHO, 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
Minimal configuration overhead	Yes	Yes	No	Yes
Support for stream processing	Yes	Lacking	No	Yes
Support for batch processing	Yes	Lacking	Yes	Yes
Focus on reliably and scalably copying data	Yes	Broader focus	No	No

Kafka Connect vs Alternative Options (3)

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

Kafka Connect for Data Movement

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

Hands-On Exercise: Using the JDBC Connector

- **In this Hands-On Exercise, you will use the JDBC Connector**
- **Please refer to the Hands-On Exercise Manual**

Kafka Connect for Data Movement

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

Chapter Summary

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

Basic Kafka Installation and Administration

Chapter 09

Course Contents

Basic Kafka Installation and Administration

- **In this chapter you will learn:**
  - The basics of how to install Kafka
  - The types of hardware that are recommended for Kafka clusters
  - How to perfom some common Kafka administrative tasks

Basic Kafka Installation and Administration

- **Kafka Installation**
- *Hardware Considerations*
- *Administering Kafka*
- *Chapter Summary*

Available Versions

- Confluent provides Kafka as Zip and Tar archives, and as packages for common Linux distributions
- 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

Installation on Ubuntu

- **To install on Ubuntu or Debian, add Confluent's public key and then install the package using `apt-get`**

```
wget -qO - http://packages.confluent.io/deb/2.0/archive.key | sudo apt-key add -

sudo add-apt-repository "deb http://packages.confluent.io/deb/2.0 stable main"

sudo apt-get update && sudo apt-get install confluent-platform-2.11.7
```

Installation on RedHat Enterprise Linux or CentOS

- **To install on RHEL, CentOS, or Fedora, first add Confluent's public key**

```
sudo rpm --import http://packages.confluent.io/rpm/2.0/archive.key
```

- **Add a file containing the repository information to your `/etc/yum/repos.d` directory**

```
[confluent-2.0]
name=Confluent repository for 2.0.x packages
baseurl=http://packages.confluent.io/rpm/2.0
gpgcheck=1
gpgkey=http://packages.confluent.io/rpm/2.0/archive.key
enabled=1
```

- **Install the package**

```
sudo yum install confluent-platform-2.11.7
```

Installing ZooKeeper

- **Confluent's distribution includes ZooKeeper 3.4.6**
- **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.host.name` 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 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`

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 Installation and Administration

- *Kafka Installation*
- **Hardware Considerations**
- *Administering Kafka*
- *Chapter Summary*

Server Specifications

- **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

Operating System and Software Requirements (1)

- **Choose the Linux server operating system you are most familiar with**
  - RedHat Enterprise Linux/CentOS and Ubuntu are the most common options
- **We recommend the `ext4` filesystem**
- **We recommend the latest release of JDK 1.8**
- **Use the G1 Garbage Collector**
- **Typical JVM options:**

```
-Xms6g -Xmx6g -XX:MetaspaceSize=96m -XX:+UseG1GC -XX:MaxGCPauseMillis=20
       -XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M
       -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
```

Operating System and Software Requirements (2)

- **Increase the number of open file handles**
  - Kafka needs a file descriptor for each open socket, index segment, and log segment

```
$ ulimit -n 100000
```

- **Disable swapping in `/etc/sysctl.conf`**

```
vm.swappiness = 0
```

Network Considerations

- **Gigabit Ethernet is sufficient for many applications**
  - 10Gb Ethernet will help for large installations
    - Particularly for inter-Broker communication
- **Avoid clusters that span datacenters**

Disk Considerations

- **Use multiple drives on the Kafka Brokers**
- **RAID10 is a good choice**
  - Much better write performance than RAID5
- **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
  - A single disk failure will result in a hard Broker failure
- **Mount disks with `noatime`**

Basic Kafka Installation and Administration

- *Kafka Installation*
- *Hardware Considerations*
- **Administering Kafka**
- *Chapter Summary*

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**
  - Replication factor of one, a single partition
- **Alternatively, you can create a topic manually**
  - This allows you to set the replication factor and number of partitions

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

- **You can also modify topics**

```
bin/kafka-topics.sh --zookeeper zk_host:port \
--alter --topic topic_name --partitions 40
```

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

Deleting a Topic

- **It is possible to delete a topic:**

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

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

Compressing Data

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

Basic Kafka Installation and Administration

- *Kafka Installation*
- *Hardware Considerations*
- *Administering Kafka*
- **Chapter Summary**

Chapter Summary

- **Kafka clusters are typically relatively small**
  - A few machines can handle large amounts of data
- **Cluster nodes do not need massive amounts of RAM**
- **Topics can be manually created, modified and deleted**
- **If network bandwidth is becoming a bottleneck, consider compressing messages**

Kafka Streams

Chapter 10

Course Contents

Kafka Streams

- **In this chapter you will learn:**
  - The motivation behind Kafka Streams
  - The features Kafka Streams provides

Kafka Streams

- **An Introduction to Kafka Streams**
- *Chapter Summary*

What Is Kafka Streams?

- Kafka Streams is a lightweight Java library for building distributed stream processing applications using Kafka
- No external dependencies, other than Kafka
- Supports event-at-a-time processing (not microbatching) with millisecond latency
- Provides a table-like model for data
- Supports windowing operations, and stateful processing including distributed joins and aggregation
- Has fault-tolerance and supports distributed processing

A Library, Not a Framework

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

Why Not Just Build Your Own?

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

An Example Kafka Streams Application (1)

```
public class WordCountLambdaExample {

  public static void main(String[] args) throws Exception {
    Properties streamsConfiguration = new Properties();
    // Job name must be unique on this Kafka cluster
    streamsConfiguration.put(StreamsConfig.JOB_ID_CONFIG, "wordcount-lambda-example");
    streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "localhost:2181");
    streamsConfiguration.put(StreamsConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    streamsConfiguration.put(StreamsConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    streamsConfiguration.put(StreamsConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    streamsConfiguration.put(StreamsConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);

    final Serializer<String> stringSerializer = new StringSerializer();
    final Deserializer<String> stringDeserializer = new StringDeserializer();
    final Serializer<Long> longSerializer = new LongSerializer();
    final Deserializer<Long> longDeserializer = new LongDeserializer();
```

An Example Kafka Streams Application (2)

```
    KStreamBuilder builder = new KStreamBuilder();

    KStream<String, String> textLines = builder.stream(stringDeserializer, stringDeserializer, "TextLinesTopic");

    KStream<String, Long> wordCounts = textLines
      .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
      .map((key, value) -> new KeyValue<>(value, value))
      .countByKey(stringSerializer, longSerializer, stringDeserializer, longDeserializer, "Counts")
      .toStream();

    wordCounts.to("WordsWithCountsTopic", stringSerializer, longSerializer);

    // Now that we have finished the definition of the processing topology we can actually run
    // it via start().  The Streams application as a whole can be launched just like any
    // normal Java application that has a main() method.
    KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
    streams.start();
  }
```

How To Get Kafka Streams

- **Kafka Streams will be release with Kafka 0.10**
- **A Tech Preview is available for download now from Confluent**

Kafka Streams

- *An Introduction to Kafka Streams*
- **Chapter Summary**

Chapter Summary

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

Conclusion

Chapter 11

Course Contents

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 Hadoop using Kafka Connect
  - Basic Kafka cluster administration
  - The basic features of Kafka Streams

Conclusion

- **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`**