Confluent Developer Training: Building Kafka Solutions

B7/801/A

Kafka Fundamentals

Chapter 3

Course Contents

Kafka Fundamentals

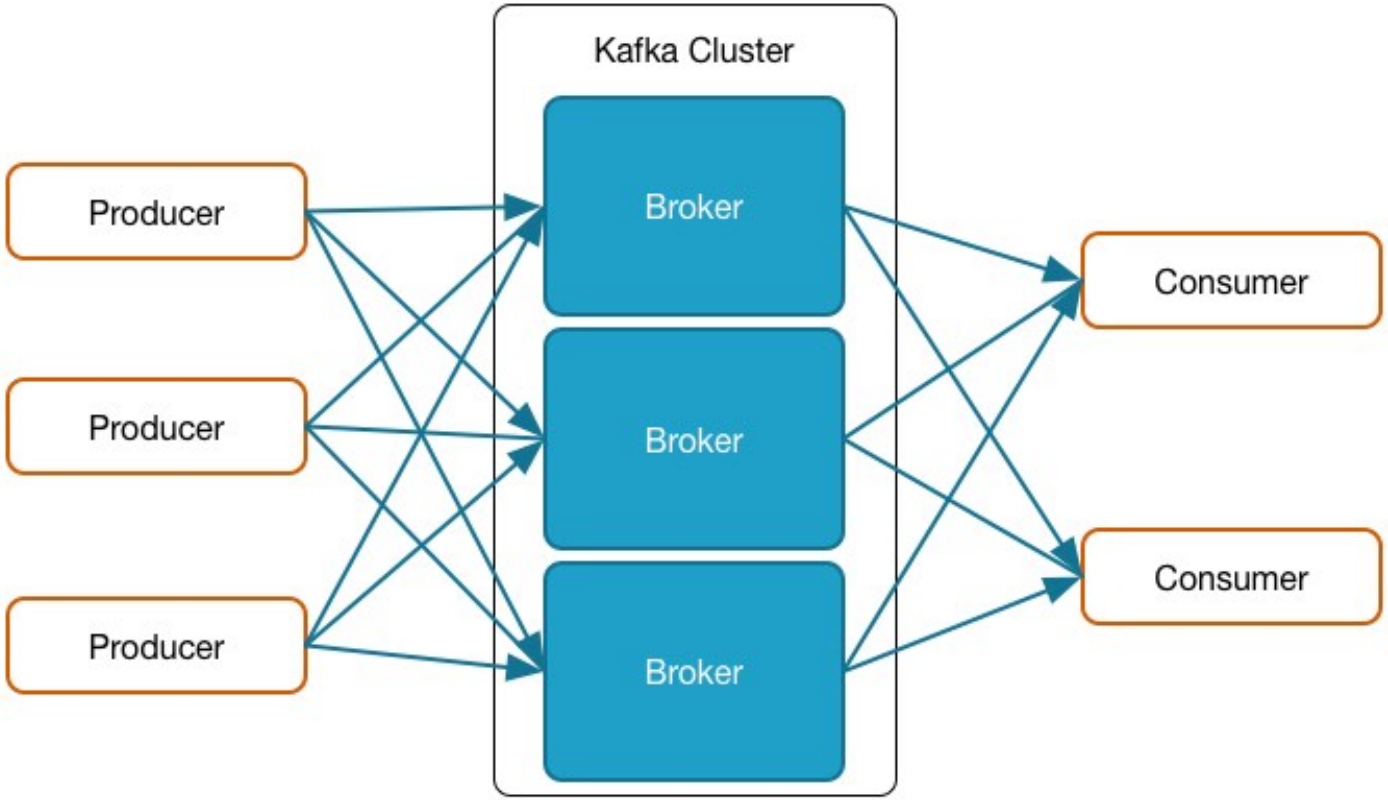
- **In this chapter you will learn:**
  - How Producers write data to a Kafka cluster
  - How data is divided into Partitions, and then stored on Brokers
  - How Consumers read data from the cluster

Kafka Fundamentals

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

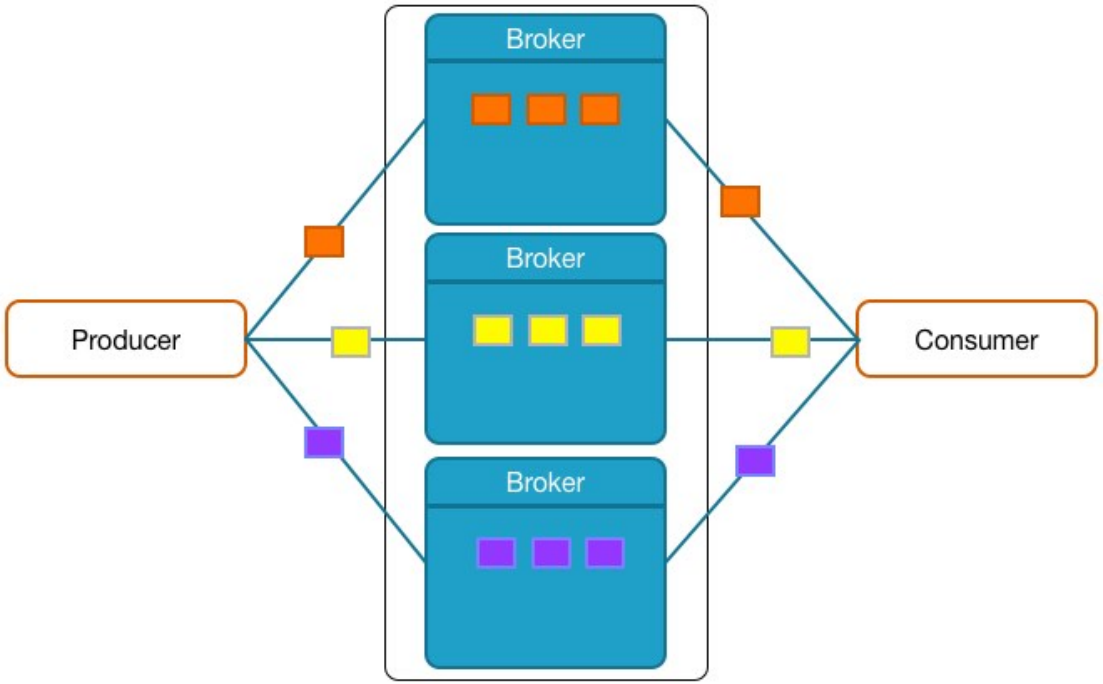
Reprise: A Very High-Level View of Kafka

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



Kafka Messages

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

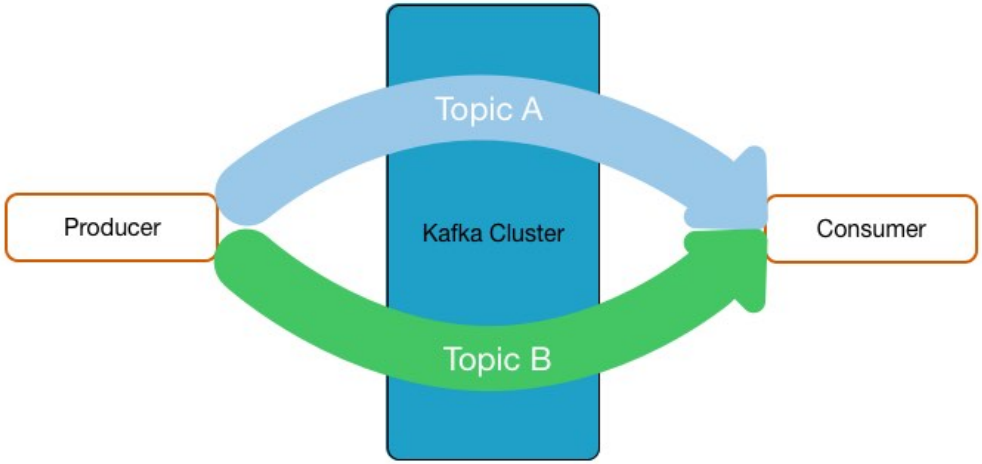


Key-Value Pairs

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

Topics

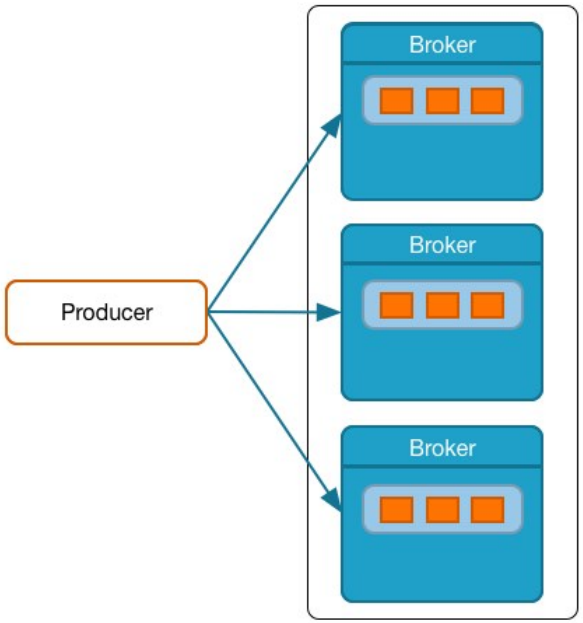
- **Kafka maintains streams of messages called *Topics***
  - Logical representation
  - They categorize messages into groups



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

Partitioned Data Ingestion

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



Load Balancing and Semantic Partitioning

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

Kafka Components

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

Kafka Fundamentals

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

Producer Basics

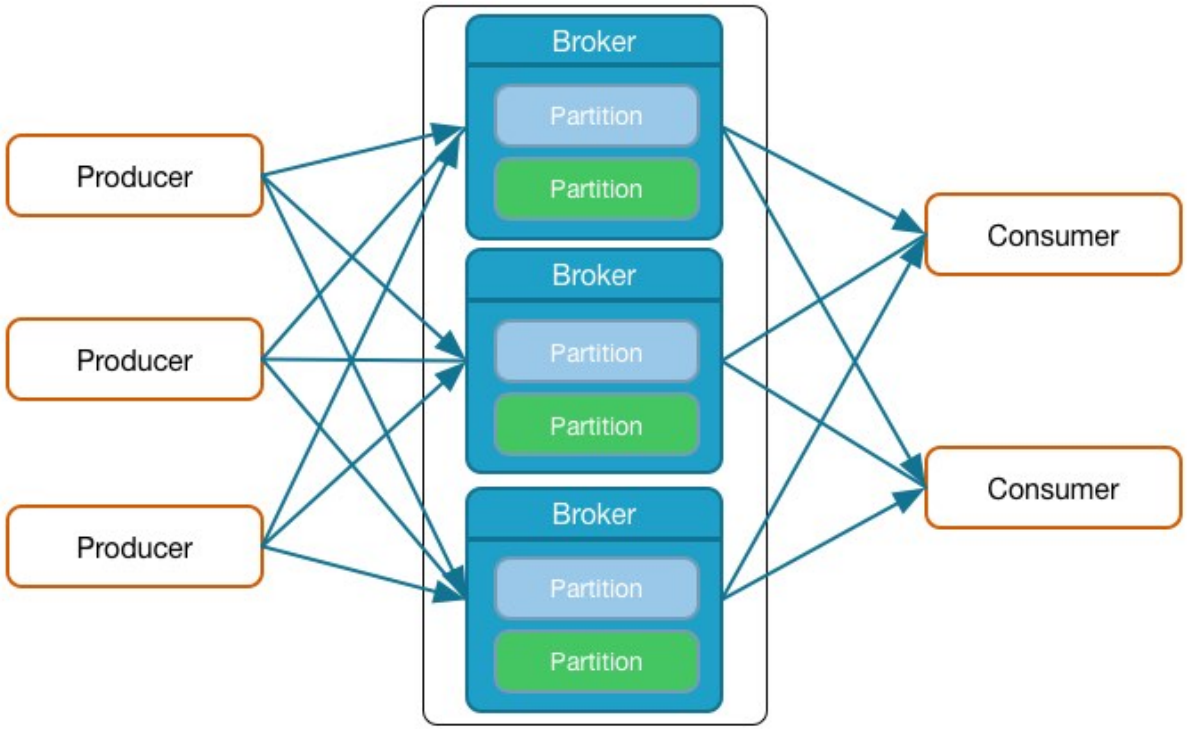
- **Producers write data in the form of *messages* to the Kafka cluster**
- **Producers can be written in any language**
  - Native Java, C/C++, Python, Go, .NET, JMS clients are supported by Confluent
  - Clients for many other languages exist
  - Confluent develops and supports a REST (REpresentational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

Kafka Fundamentals

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

Broker Basics

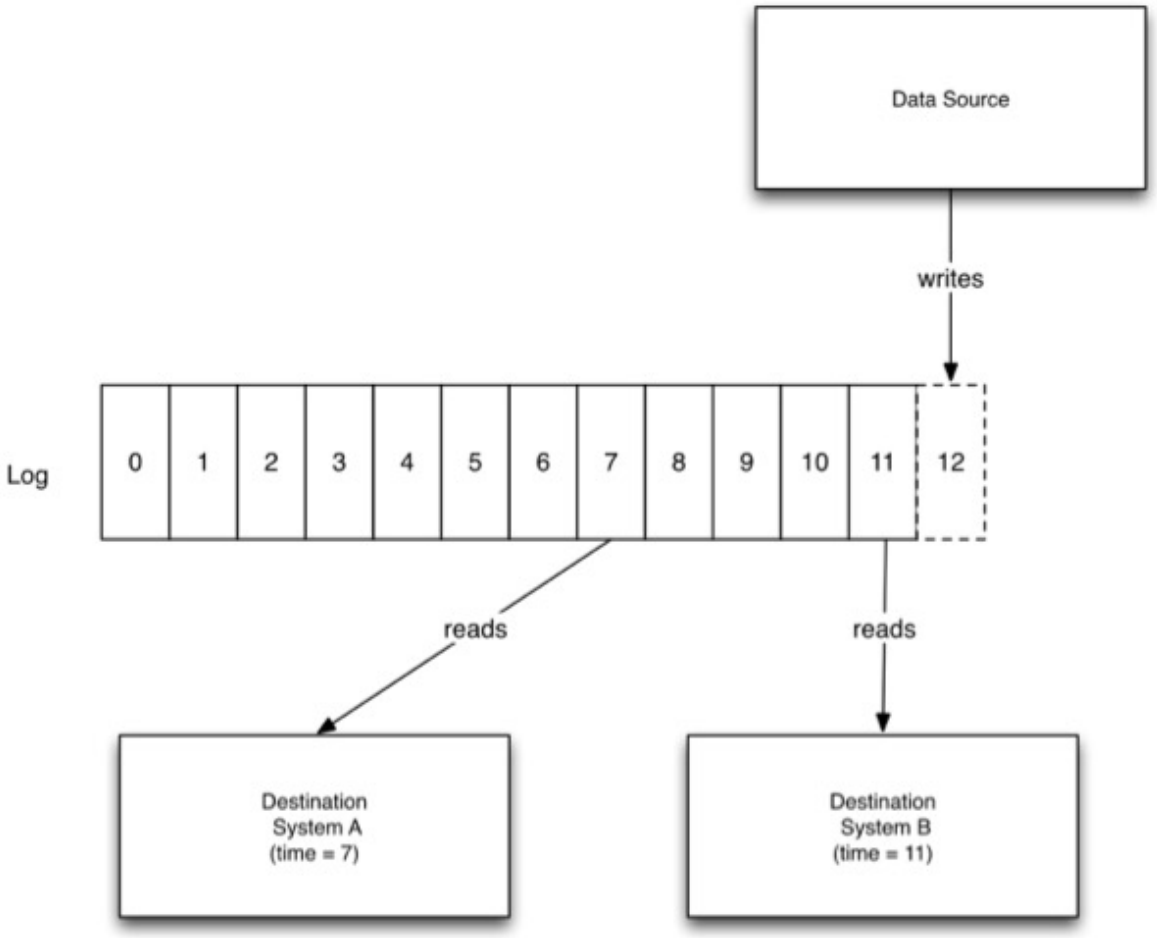
- **Brokers receive and store messages when they are sent by the Producers**
- **A production Kafka cluster will have three or more Brokers**
  - Each can handle hundreds of thousands, or millions, of messages per second



Brokers Manage Partitions

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

Messages are Stored in a Persistent Log

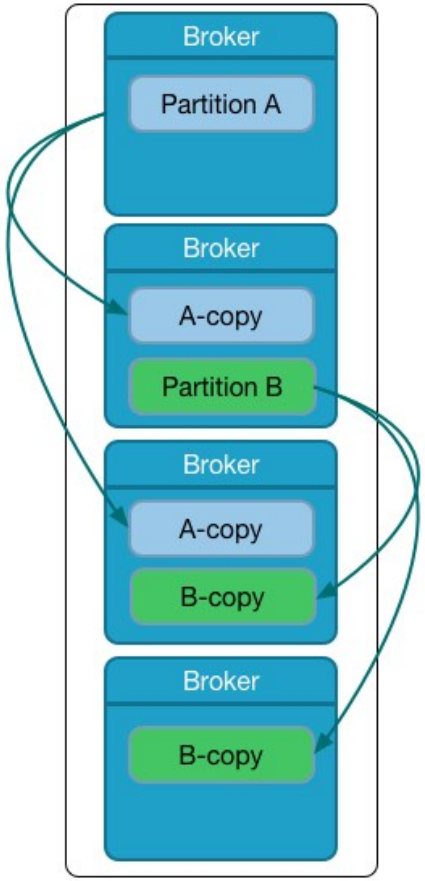


Metadata in a Kafka Message

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

Fault Tolerance via a Replicated Log

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



Kafka Fundamentals

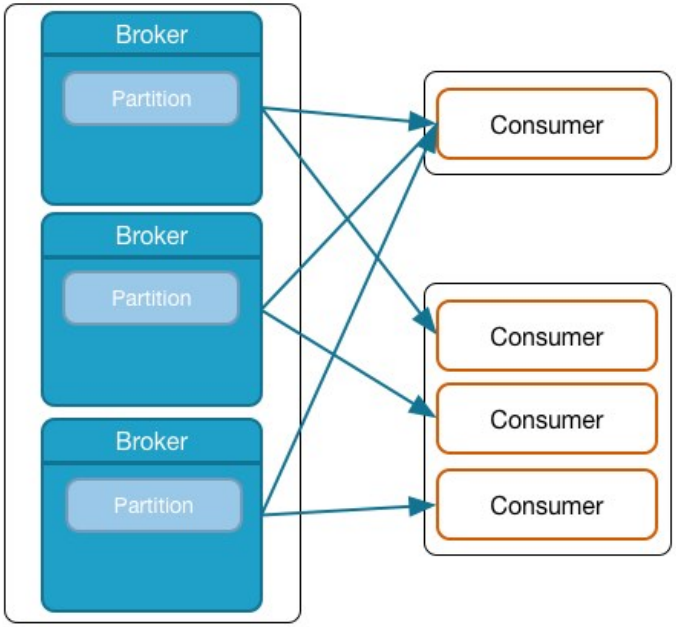
- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- **Kafka Consumers**
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

Consumer Basics

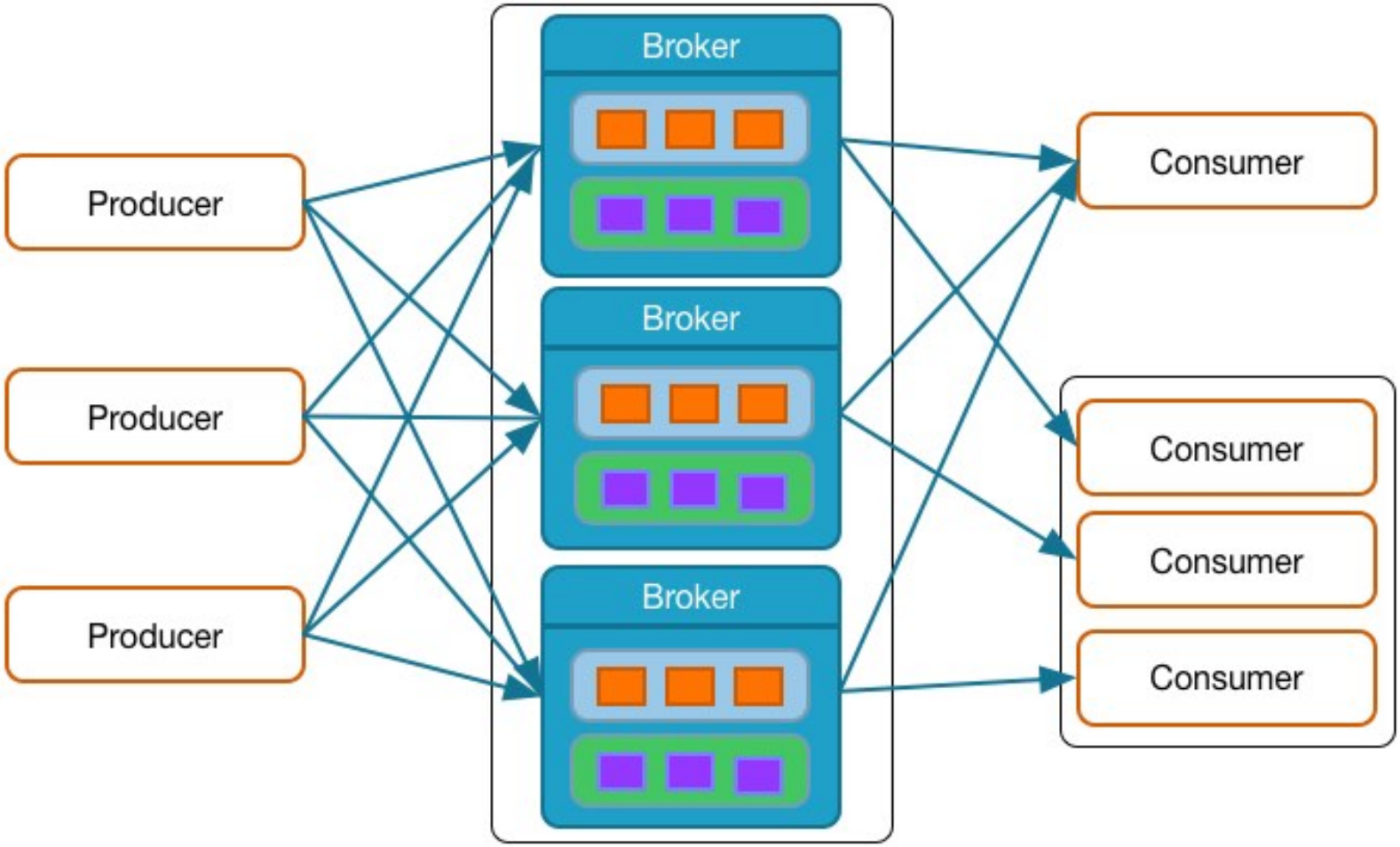
- **Consumers pull messages from one or more Topics in the cluster**
  - As messages are written to a Topic, the Consumer will automatically retrieve them
- **The *Consumer Offset* keeps track of the latest message read**
  - If necessary, the Consumer Offset can be changed
    - For example, to reread messages
- **The Consumer Offset is stored in a special Kafka Topic**
- **A command-line Consumer tool exists to read messages from the cluster**
  - Useful for testing, debugging, etc.

Distributed Consumption

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



The Result: A Linearly Scalable Firehose



Kafka Fundamentals

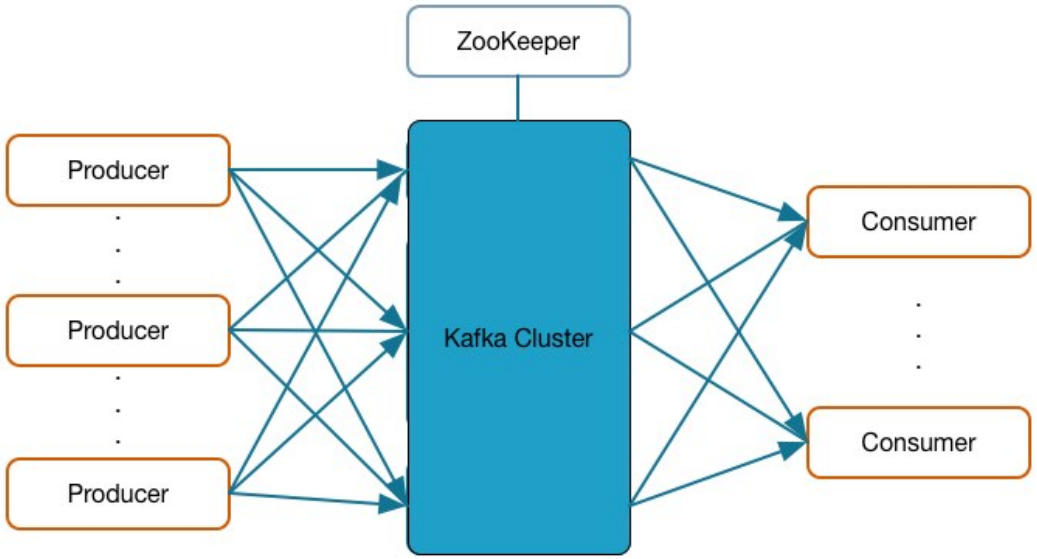
- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- **Kafka's Use of ZooKeeper**
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

What is ZooKeeper?

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

How Kafka Uses ZooKeeper

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



Quiz: Question

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

Quiz: Answer

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

Kafka Fundamentals

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

Kafka's Benefits over Traditional Message Queues

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

Multi-Subscription and Scalability

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

The Advantages of a Pull Architecture

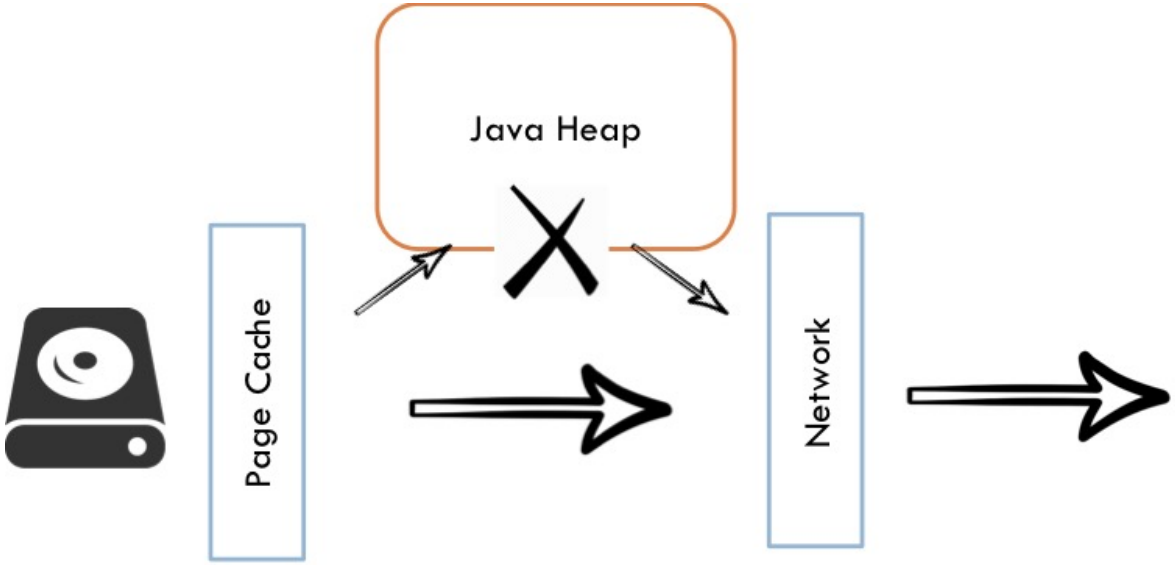
- **Kafka Consumers pull messages from the Brokers**
  - This is in contrast to some other systems, which use a *push* design
- **The advantages of pulling, rather than pushing, data, include:**
  - The ability to add more Consumers to the system without reconfiguring the cluster
  - The ability for a Consumer to go offline and return later, resuming from where it left off
  - No problems with the Consumer being overwhelmed by data
    - It can pull, and process, the data at whatever speed it needs to
    - A slow Consumer will not affect Producers

The Page Cache for High Performance

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

Speeding Up Data Transfer

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



Kafka Fundamentals

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

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

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

Kafka Fundamentals

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

Chapter Review

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

Kafka’s Architecture

Chapter 4

Course Contents

Kafka's Architecture

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

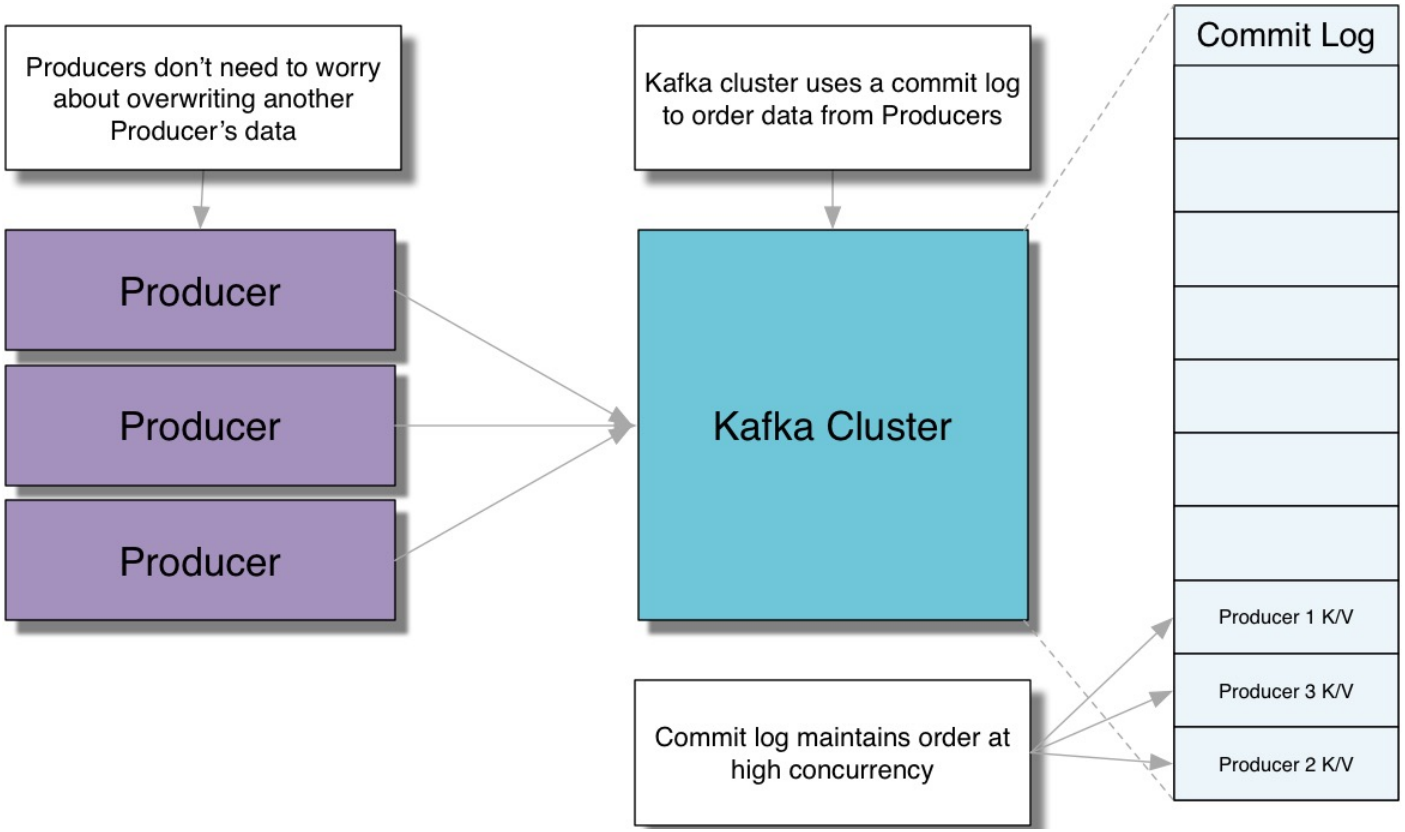
Kafka's Architecture

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

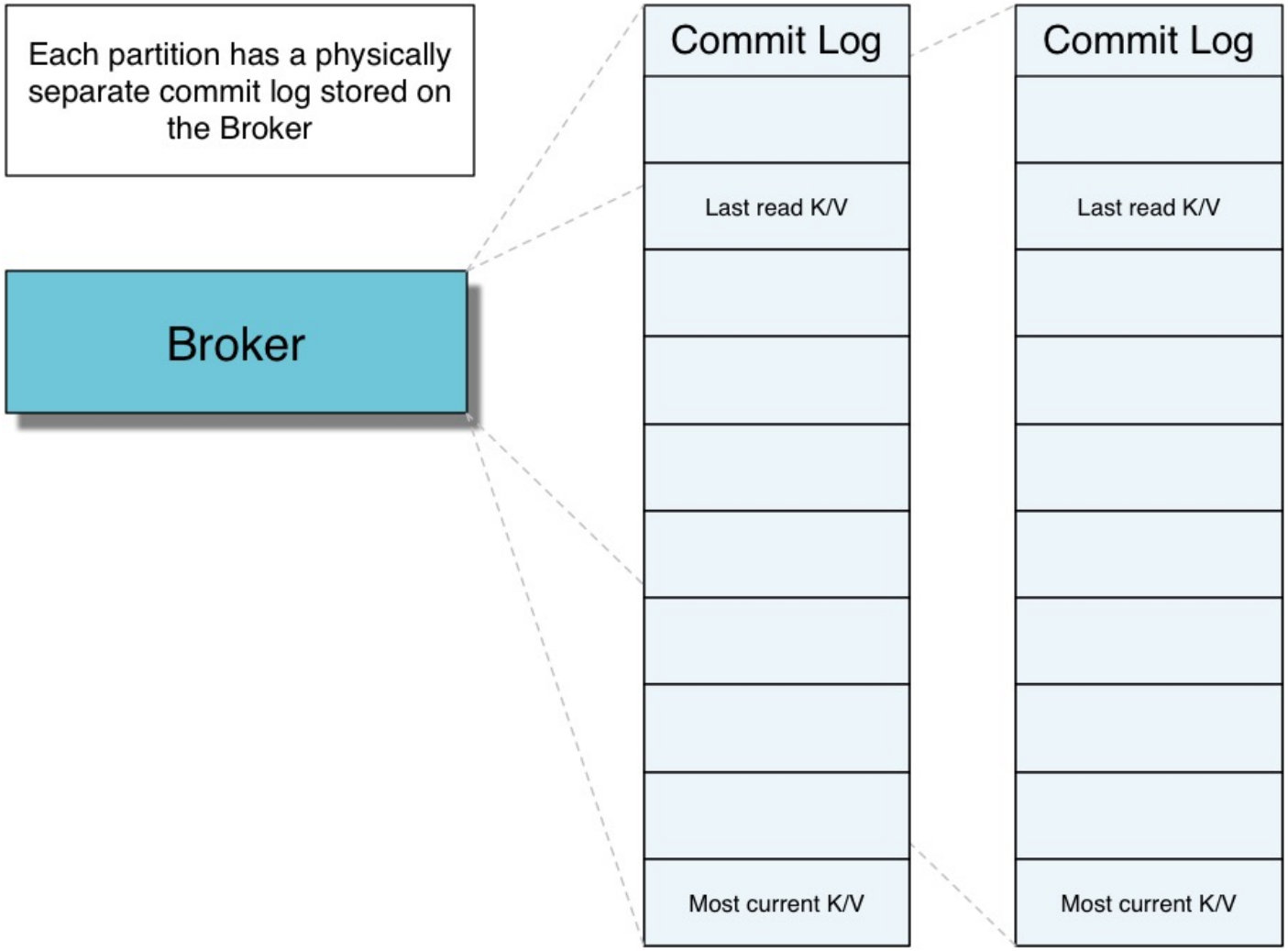
What is a Commit Log?

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

The Commit Log for High Concurrency



Partitions Are Stored as Separate Logs



Kafka's Architecture

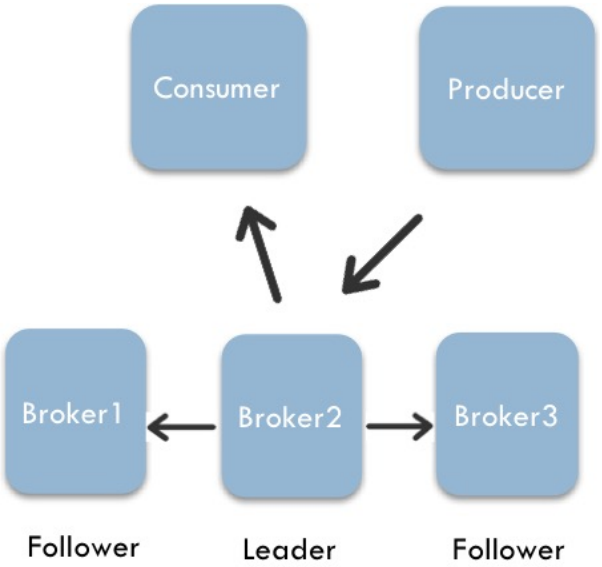
- *Kafka's Log Files*
- **Replicas for Reliability**
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Review*

Problems With our Current Model

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

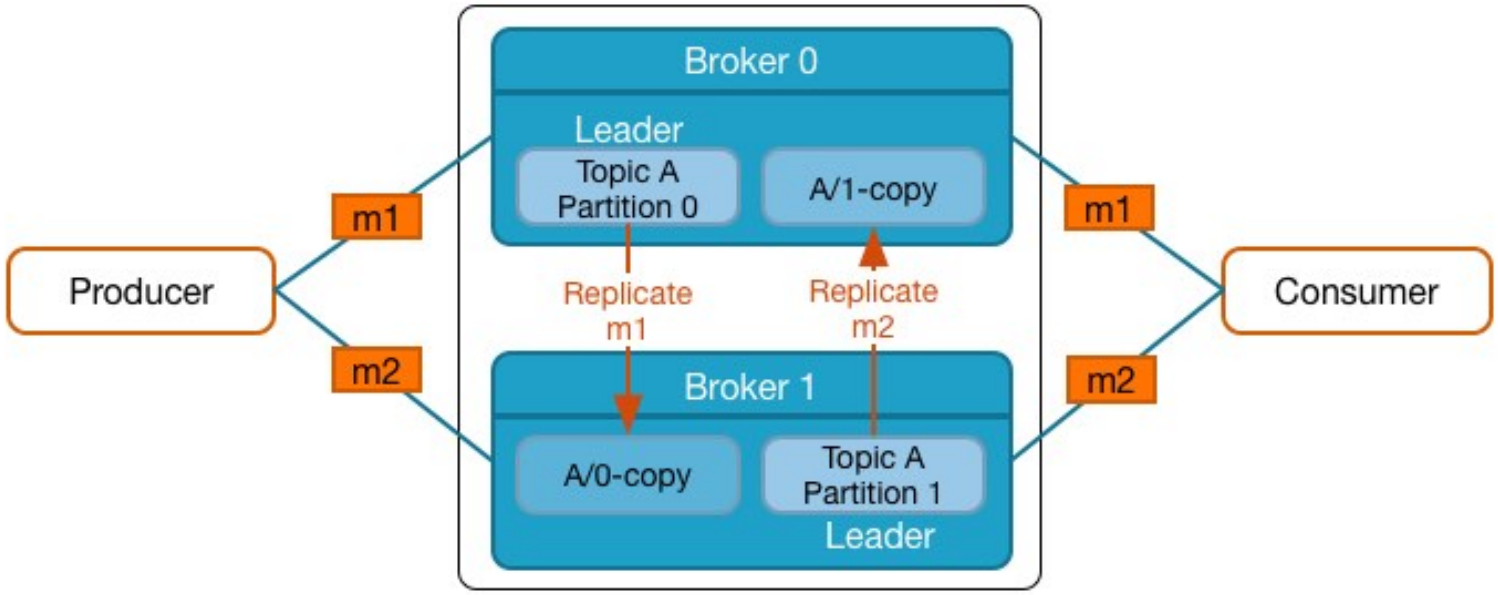
Replication of Partitions

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



Important: Clients Do Not Access Followers

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



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

In-Sync Replicas

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

Managing Broker Failures

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

Kafka's Architecture

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

Scaling using Partitions

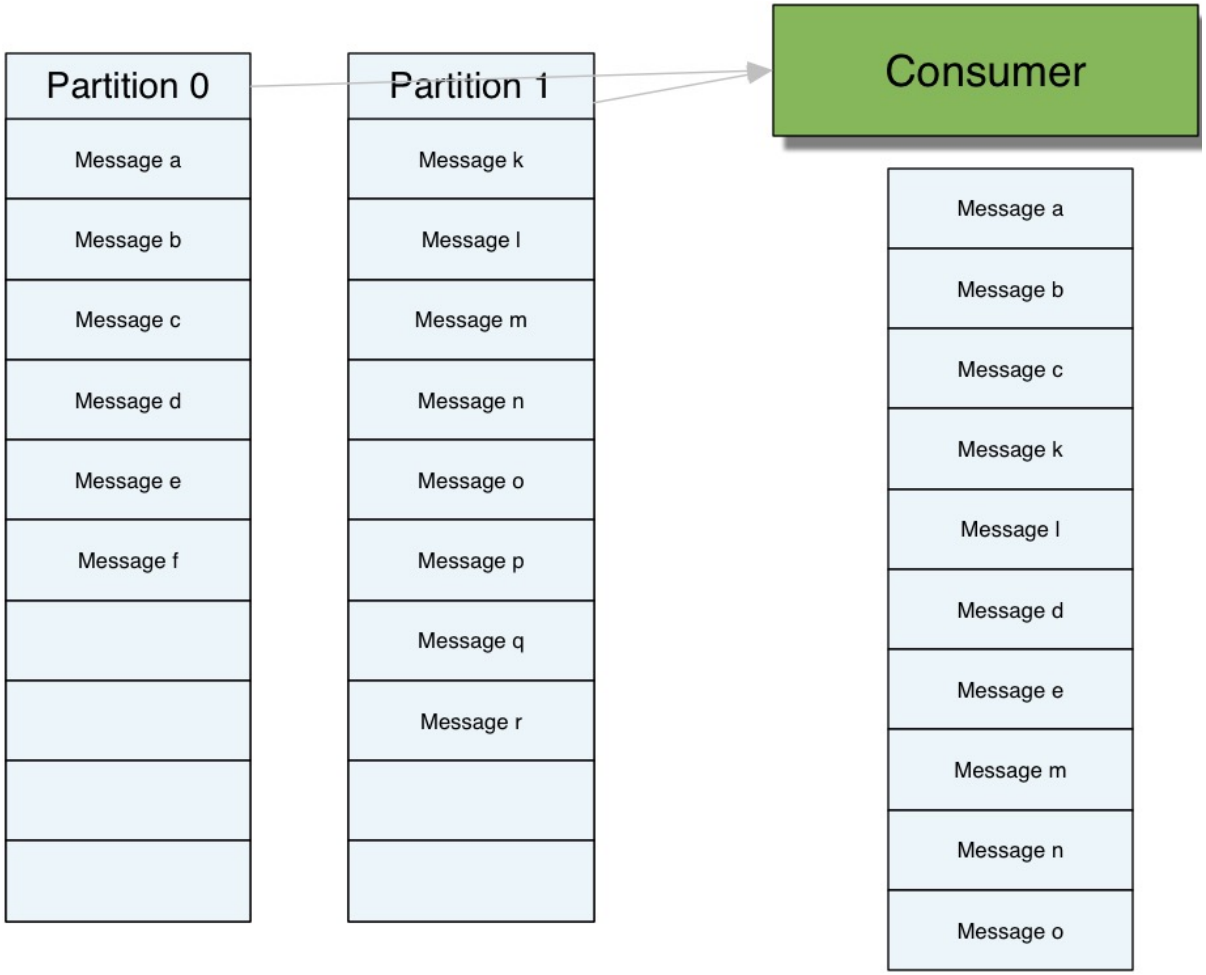
- **Recall: All Consumers read from the leader of a Partition**
  - No clients write to, or read from, followers
- **This can lead to congestion on a Broker if there are many Consumers**
- **Splitting a Topic into multiple Partitions can help to improve performance**
  - Leaders for different Partitions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

Preserve Message Ordering

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

An Important Note About Ordering

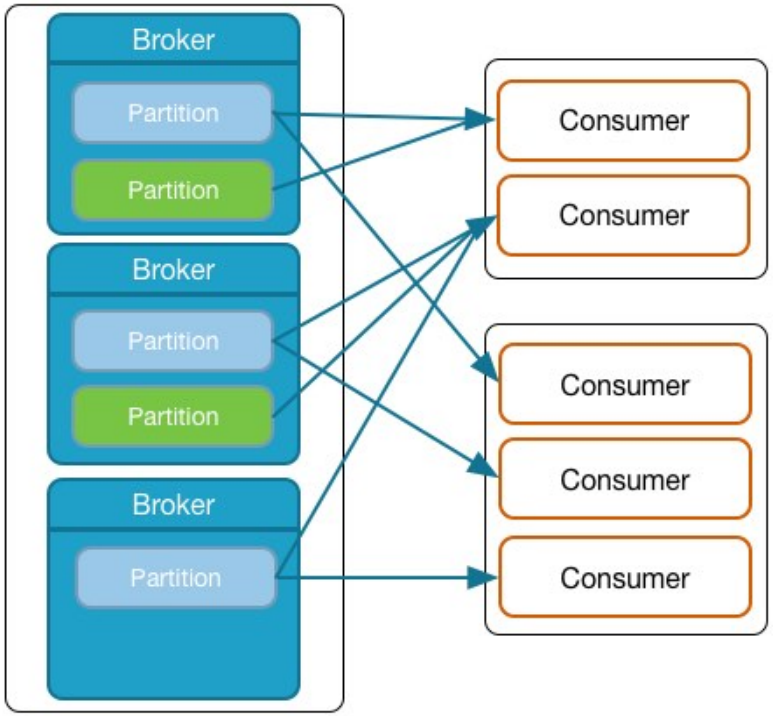
- **If there are multiple Partitions, you** will not get total ordering across all messages **when reading data**



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

Group Consumption Balances Load

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

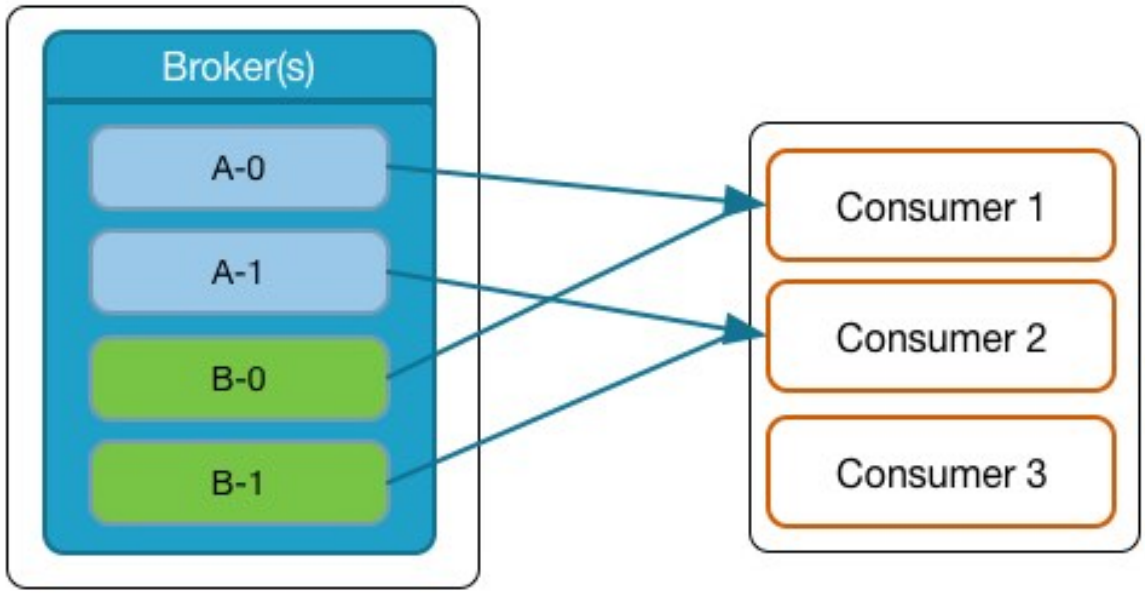


Partition Assignment within a Consumer Group

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

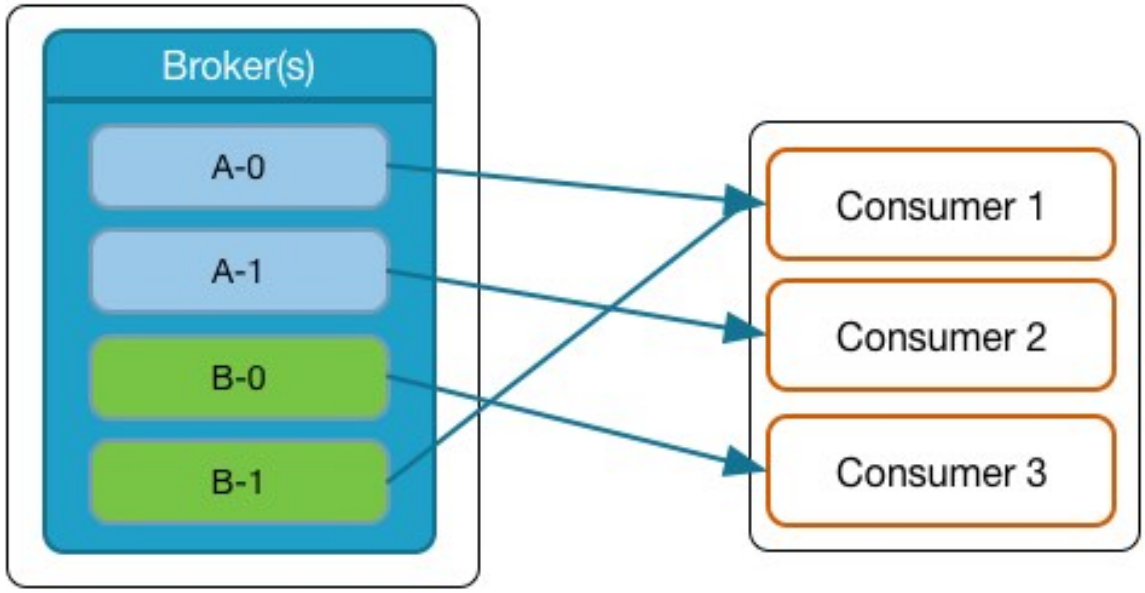
Partition Assignment Strategy: Range

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



Partition Assignment Strategy: RoundRobin

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



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

Partition Assignment Strategy: Sticky

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

Consumer Groups: Limitations

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

Consumer Groups: Caution When Changing Partitions

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

Kafka's Architecture

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

Hands-On Exercise: Consuming from Multiple Partitions

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

Kafka's Architecture

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

Chapter Review

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

Developing With Kafka

Chapter 5

Course Contents

Developing With Kafka

- **In this chapter you will learn:**
  - How to write a Producer using the Java API
  - How to use the REST proxy to access Kafka from other languages
  - How to write a basic Consumer using the New Consumer API

Developing With Kafka

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

The Kafka API

- **Since Kafka 0.9, Kafka includes Java clients in the `org.apache.kafka.clients` package**
  - These are intended to supplant the older Scala clients
  - They are available in a JAR which has as few dependencies as possible, to reduce code size
- **There are client libraries for many other languages**
  - The quality and support for these varies
- **Confluent provides and supports client libraries for C/C++, Python, Go, and .NET**
- **Confluent also maintains a REST Proxy for Kafka**
  - This allows any language to access Kafka via REST
    - (REpresentational State Transfer; essentially, a way to access a system by making HTTP calls)

Our Class Environment

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

Developing With Kafka

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

The Producer API

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

Important **`Properties`** Elements (1)

Name	Description
`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)

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

Creating the **Properties** and **KafkaProducer** Objects

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

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

Helper Classes

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

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

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

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

Sending Messages to Kafka

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

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

② **Alternatively:**

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

`send()` Does Not Block

- **The `send()` call is asynchronous**
  - It does not block; it returns immediately and your code continues
- **It returns a `Future` which contains a `RecordMetadata` object**
  - The Partition the record was put into and its offset
- **To force `send()` to block, call `producer.send(record).get()`**

When Do Producers Actually Send?

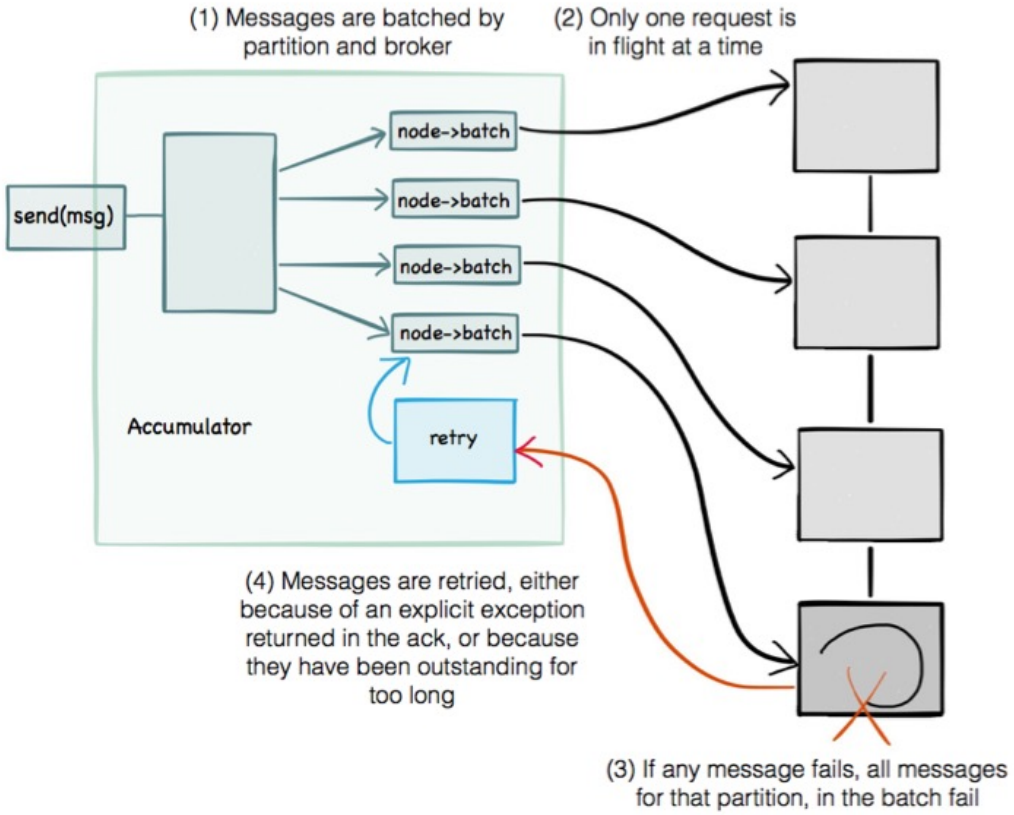
- **A Producer `send()` returns immediately after it has added the message to a local buffer of pending record sends**
  - This allows the Producer to send records in batches for better performance
- **Then the Producer flushes multiple messages to the Brokers based on batching configuration parameters**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

Producer Retries

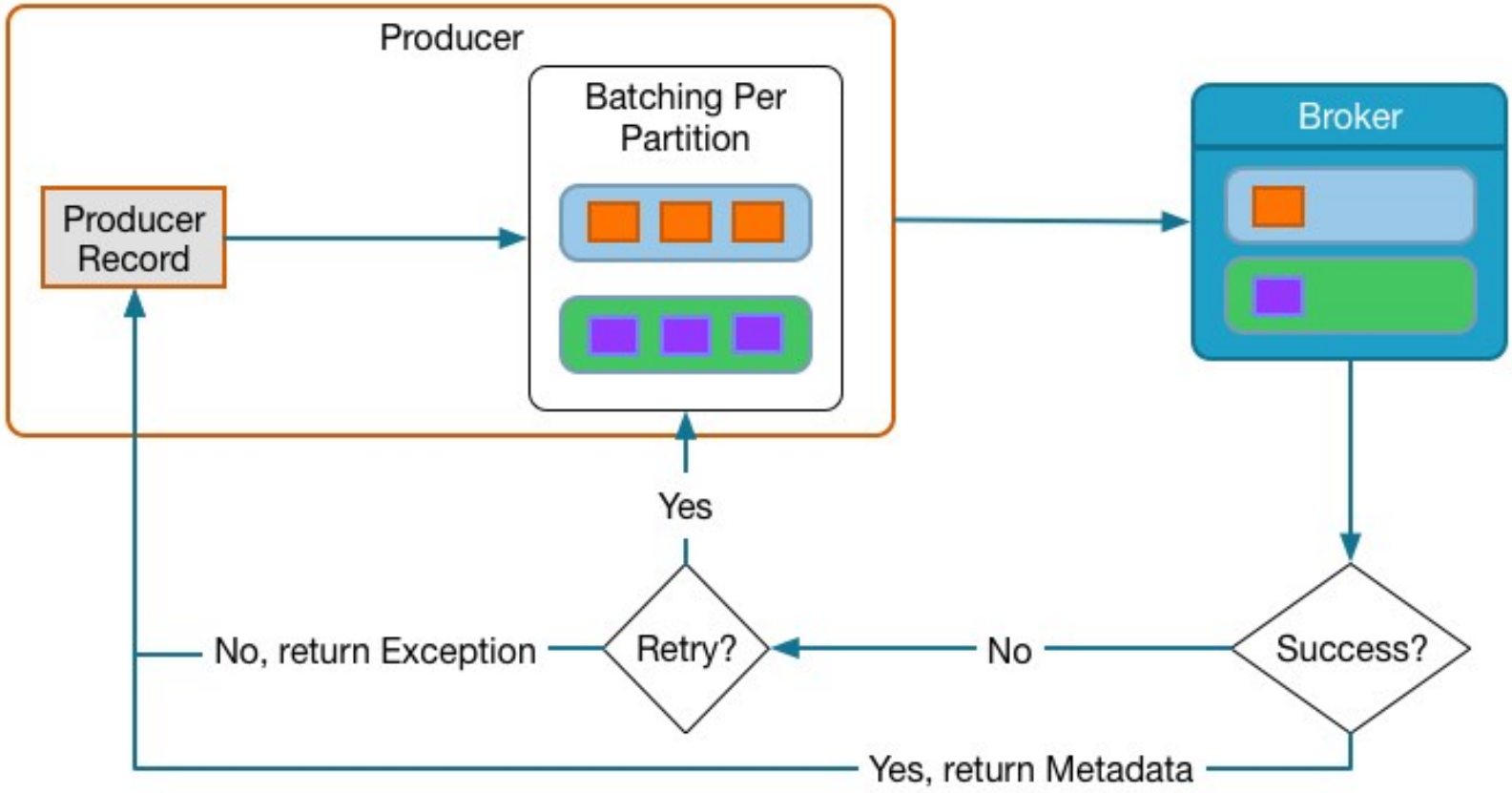
- **Developers can configure the `retries` configuration setting in the Producer code**
  - `retries`: how many times the Producer will attempt to retry sending records
    - Only relevant if `acks` is not `0`
  - Hides transient failure
  - Ensure lost messages are retried rather than just throwing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

Preserve Message Send Order

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



Batching and Retries



Performance Tuning Batching

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

`send()` and Callbacks (1)

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

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

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

`send()` and Callbacks (2)

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

```
producer.send(record, (recordMetadata, e) -> {
  if (e != null) {
    e.printStackTrace();
  } else {
    System.out.println("Message String = " + record.value() + ", Offset = " + recordMetadata
.offset());
  }
});
```

Closing the Producer

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

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

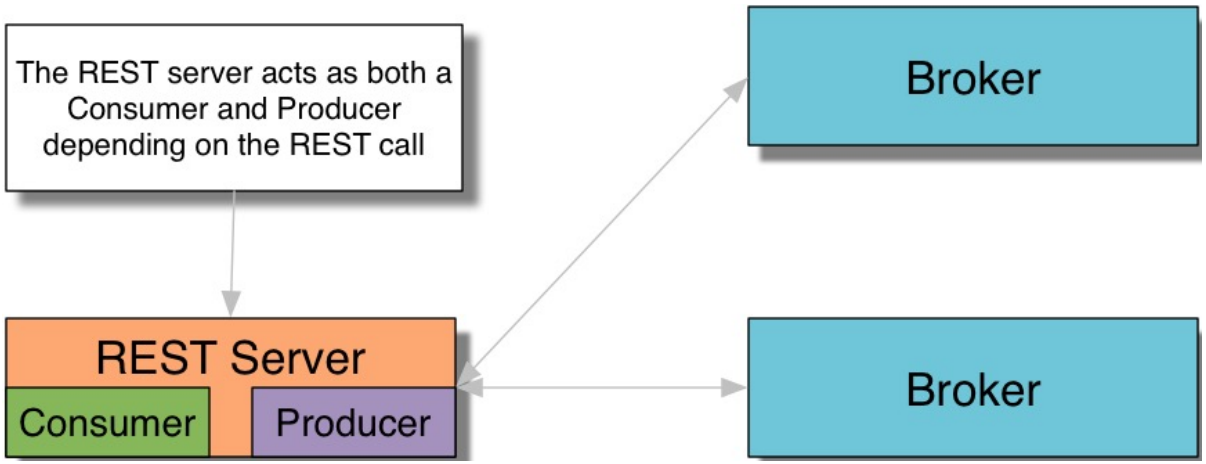
- **`close(timeout, timeUnit)`: waits up to timeout for the producer to complete the sending of all incomplete requests**
  - If the producer is unable to complete all requests before the timeout expires, this method will fail any unsent and unacknowledged records immediately

Developing With Kafka

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

About the REST Proxy

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



A Python Producer Using the REST Proxy

```
#!/usr/bin/python

import requests
import base64
import json

url = "http://restproxy: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

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

Hands-On Exercise: Writing a Producer

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

Developing With Kafka

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

Consumers and Offsets

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

Important Consumer Properties

- **Important Consumer properties include:**

Name	Description
`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

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

① **The Consumer can subscribe to as many Topics as it wishes. Note that this call is not additive; calling `subscribe` again will remove the existing list of Topics, and will only subscribe to those specified in the new call**

② **You may also use regular expressions (*i.e.*, `Pattern`) for topic subscription**

Reading Messages from Kafka with `poll()`

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

① **Loop forever**

② **Each call to `poll` returns a (possibly empty) list of messages.**

Controlling the Number of Messages Returned

- **By default, `poll()` fetches available messages from multiple partitions across multiple Brokers**
  - Up to the maximum data size *per partition*
    - `max.partition.fetch.bytes`: default value `1048576`
  - A Topic with many partitions could result in extremely large amounts of data being returned
- **The optional timeout parameter in the call to `poll()` is across all fetch requests**
  - This controls the maximum amount of time in milliseconds that the Consumer will block if no new records are available
  - If records are available, it will return immediately
- **In Kafka 0.10.0, `max.poll.records` was introduced**
  - Property limits the total number of records retrieved in a single call to `poll()`

Performance Tuning Consumer Fetch Requests

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

Message Size Limit

- **Try not to change the maximum message size unless it is unavoidable**
  - Kafka is not optimized for very large messages
- **If you must change the maximum size for a batch of messages that the Broker can receive from a Producer**
  - Broker: `message.max.bytes` (Default: 1MB)
  - Topic override: `max.message.bytes` (Default: 1MB)

Preventing Resource Leaks

- It is good practice to wrap the code in a `try{ }` block, and close the `KafkaConsumer` object in a `finally{ }` block to avoid resource leaks
- It is important to note that `KafkaConsumer` is not thread-safe

```
1 try {
2   while (true) { ①
3     ConsumerRecords<String, String> records = consumer.poll(100); ②
4     for (ConsumerRecord<String, String> record : records)
5       System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(),
record.value());
6   }
7 } finally {
8   consumer.close();
9 }
```

Developing With Kafka

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

A Python Consumer Using the REST API (1)

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

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

A Python Consumer Using the REST API (2)

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

A Python Consumer Using the REST API (3)

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

A Python Consumer Using the REST API (4)

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

Developing With Kafka

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

Hands-On Exercise: Writing a Basic Consumer

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

Developing With Kafka

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

Chapter Review

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

More Advanced Kafka Development

Chapter 6

Course Contents

More Advanced Kafka Development

- **In this chapter you will learn:**
  - How to enable exactly once semantics (EOS)
  - How to specify the offset to read from
  - How to manually commit reads from the Consumer
  - How to create a custom Partitioner
  - How to control message delivery reliability

More Advanced Kafka Development

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

Motivation for Exactly Once Semantics

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

EOS in Kafka

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

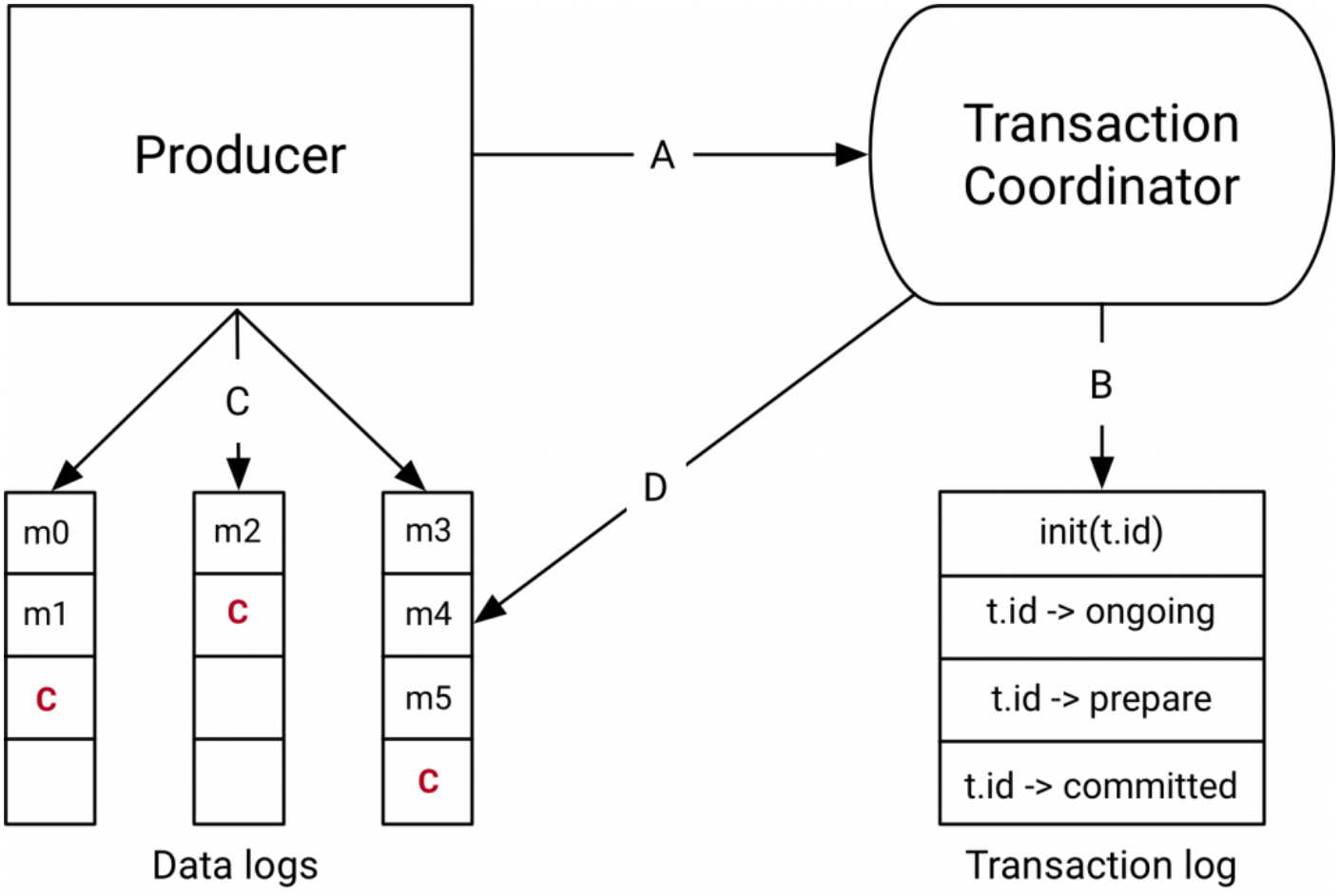
Enabling EOS on the Producer Side (1)

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

Enabling EOS on the Producer Side (2)

```
producer.initTransactions();
try {
  producer.beginTransaction();
  producer.send(record1);
  producer.send(record2);
  producer.commitTransaction();
} catch(ProducerFencedException e) {
  producer.close();
} catch(KafkaException e) {
  producer.abortTransaction();
}
```

Producer Transactions across Partitions



Enabling EOS on the Consumer Side

- **Configure the Consumer to read committed transactional messages**
  - Set `isolation.level` to `read_committed` (Default: `read_uncommitted`)
    - `read_committed`: reads only committed transactional messages and all non-transactional messages
    - `read_uncommitted`: reads all transactional messages and all non-transactional messages

More Advanced Kafka Development

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

Determining the Offset When a Consumer Starts

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

Changing the Offset Within the Consumer (1)

- **The `KafkaConsumer` API provides ways to view offsets and dynamically change the offset from which the Consumer will read**
- **View offsets**
  - `position(TopicPartition)` provides the offset of the next record that will be fetched
  - `offsetsForTimes(Map<TopicPartition,Long> timestampsToSearch)` looks up the offsets for the given Partitions by timestamp
- **Change offsets**
  - `seekToBeginning(Collection<TopicPartition>)` seeks to the first offset of each of the specified Partitions
  - `seekToEnd(Collection<TopicPartition>)` seeks to the last offset of each of the specified Partitions
  - `seek(TopicPartition, offset)` seeks to a specific offset in the specified Partition

Changing the Offset Within the Consumer (2)

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

```
1 consumer.subscribe(Arrays.asList("my_topic"));
2 consumer.poll(0);
3 consumer.seekToBeginning(consumer.assignment()); ①
```

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

Changing the Offset Within the Consumer (3)

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

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

① Add each Partition and timestamp to the HashMap

② Get the offsets for each Partition

③ Seek to the specified offset for each Partition

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

More Advanced Kafka Development

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

Consumer ‘Liveness’

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

Consumer Heartbeats for 'Liveness'

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

Pausing Record Retrieval

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

Partition Assignments for Consumers

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

Consumer Rebalancing

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

The Case For and Against Rebalancing

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

More Advanced Kafka Development

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

Default Behavior: Automatically Committed Offsets

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

Problems With Automatically Committed Offsets

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

Manually Committing Offsets

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

Manually Committing Offsets From the REST Proxy

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

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

Aside: What Offset is Committed?

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

Storing Offsets Outside of Kafka

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

Quiz: Question

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

Quiz: Answer

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

More Advanced Kafka Development

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

Kafka's Default Partitioning Scheme

- **Recall that by default, if a message has a key, Kafka will hash the key and use the result to map the message to a specific Partition**
- **This means that all messages with the same key will go to the same Partition**
- **If the key is null and the default Partitioner is used, the record will be sent to a random Partition (using a round-robin algorithm)**
- **You may wish to override this behavior and provide your own Partitioning scheme**
  - For example, if you will have many messages with a particular key, you may want one Partition to hold just those messages

Creating a Custom Partitioner

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

Custom Partitioner: Example

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

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

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

An Alternative to a Custom Partitioner

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

More Advanced Kafka Development

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

Replication Factor Affects Message Durability

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

Delivery Acknowledgment

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

`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 acknowledgment from all in-sync replicas before reporting the message as delivered to the Producer

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

Important Timeouts

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

More Advanced Kafka Development

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

Hands-On Exercise: Accessing Previous Data

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

More Advanced Kafka Development

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

Chapter Review

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

Schema Management In Kafka

Chapter 7

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 Review*

What is Serialization?

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

The Need for a More Complex Serialization System

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

Avro: An Efficient Data Serialization System

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

Schema Management In Kafka

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

Avro Schemas

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

Avro Data Types (Simple)

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

Name	Description	Java equivalent
`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
`record`	A user-defined field comprising one or more simple or complex data types, including nested records
`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` is the most important of these, as we will see**

Example Avro Schema (1)

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

Example Avro Schema (2)

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

Example Schema Snippet with **array** and **map**

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

Example Schema Snippet with **enum**

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

Schema Evolution

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

Compatibility Examples

- **Consider a schema written with the following fields**

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

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

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

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

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

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

Schema Management In Kafka

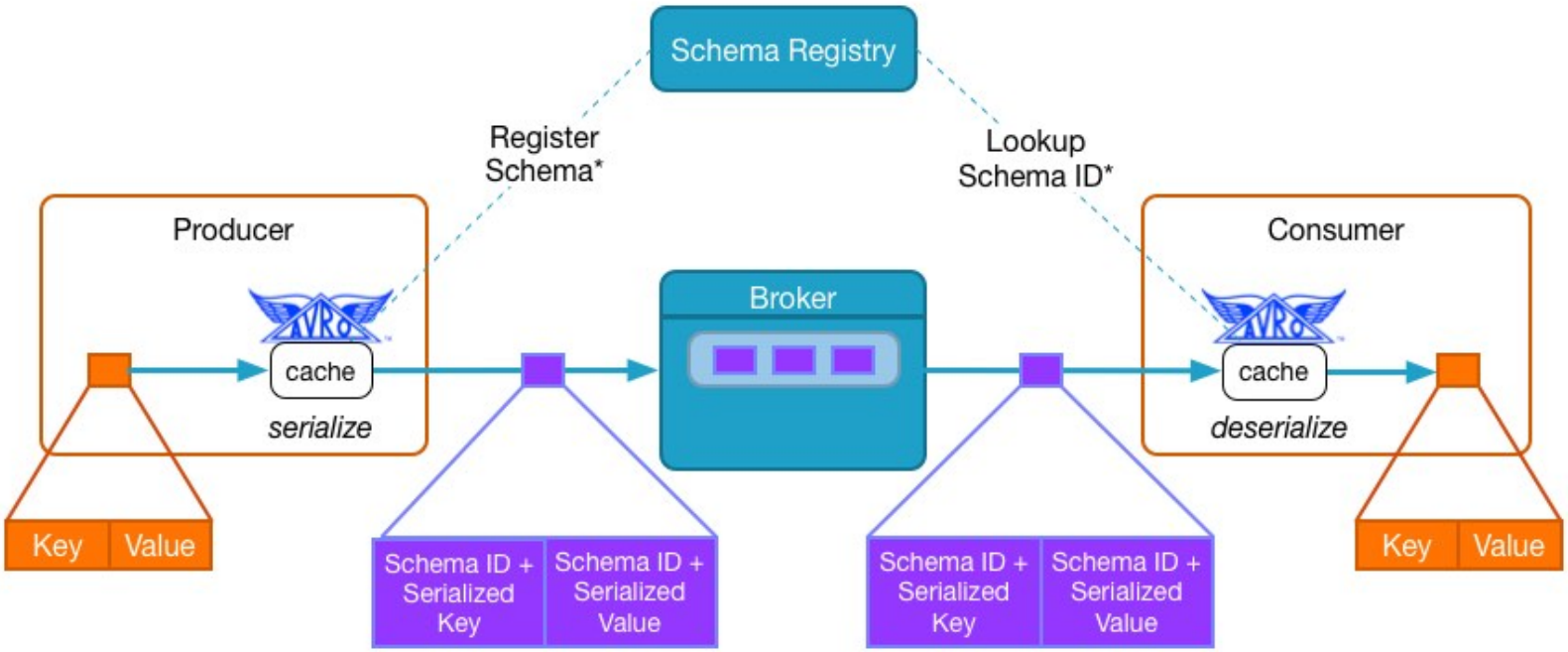
- *An Introduction to Avro*
- *Avro Schemas*
- **The Schema Registry**
- *Hands-On Exercise: Using Kafka with Avro*
- *Chapter Review*

What Is the Schema Registry?

- **Schema registry provides centralized management of schemas**
  - Stores a versioned history of all schemas
  - Provides a RESTful interface for storing and retrieving Avro schemas
  - Checks schemas and throws an exception if data does not conform to the schema
  - Allows evolution of schemas according to the configured compatibility setting
- **Sending the Avro schema with each message would be inefficient**
  - Instead, a globally unique ID representing the Avro schema is sent with each message
- **The Schema Registry stores schema information in a special Kafka topic**
- **The Schema Registry is accessible both via a REST API and a Java API**
  - There are also command-line tools, `kafka-avro-console-producer` and `kafka-avro-console-consumer`

Schema Registration and Dataflow

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



Client Support for the Schema Registry (1)

- **Java client**
  - New schemas automatically registered

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

- **Python client (`confluent-kafka-python`)**
  - Support for Schema Registry introduced in Kafka 0.10.2 (Confluent 3.2)

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

Client Support for the Schema Registry (2)

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

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

Different Versions of Schemas in the Same Topic

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

```
$ curl -X PUT -i -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data '{"compatibility": "NONE"}' \
    http://schemaregistry1:8081/config/my_topic
```

Java Avro Producer example

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "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(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"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

```
 1 public class CardConsumer {
 2   public static void main(String[] args) {
 3     Properties props = new Properties();
 4     props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
 5     props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
 6     props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
 7     props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
 8     props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://schemaregistry1:8081");
 9     props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
10
11     KafkaConsumer<CardSuit, SimpleCard> consumer = new KafkaConsumer<>(props);
12     consumer.subscribe(Arrays.asList("my_avro_topic"));
13
14     while (true) {
15       ConsumerRecords<CardSuit, SimpleCard> records = consumer.poll(100);
16         for (ConsumerRecord<CardSuit, SimpleCard> record : records) {
17           System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.
key().getSuit(), record.value().getCard());
18
19         }
20     }
21   }
22 }
```

Python Avro Producer Example

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

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 \
--from-beginning --topic my_avro_topic
```

Schema Management In Kafka

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

Hands-On Exercise: Using Kafka with Avro

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

Schema Management In Kafka

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

Chapter Review

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