

Advanced Apache Kafka

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or professional environment.



Table of Contents

1. Introduction to Messaging Systems: 4
 2. Introducing Kafka, the Distributed Messaging Platform: 24
 3. Deep Dive into Kafka Producers: 47
 4. Deep Dive into Kafka Consumers: 75
 5. Building Spark Streaming Applications with Kafka: 99
 6. Building Storm Applications with Kafka: 125
 7. Using Kafka with Confluent Platform: 162
- 



Table of Contents

8. Building ETL Pipelines Using Kafka: 196
 9. Building Streaming Applications Using Kafka Streams: 217
 10. Kafka Cluster Deployment: 243
 11. Using Kafka in Big Data Applications: 267
 12. Securing Kafka: 295
 13. Streaming Application Design Considerations: 338
- 

1. Introduction to Messaging Systems

A blurred background image of a person's hands typing on a laptop keyboard, set against a dark grey diagonal overlay.

Introduction to Messaging Systems

We will be covering the following topics in this lesson:

- Principles of designing a good messaging system
- How a messaging system works
- A point-to-point messaging system
- A publish-subscribe messaging system
- The AMQP messaging protocol
- Finally we will go through the messaging system needed in designing streaming applications

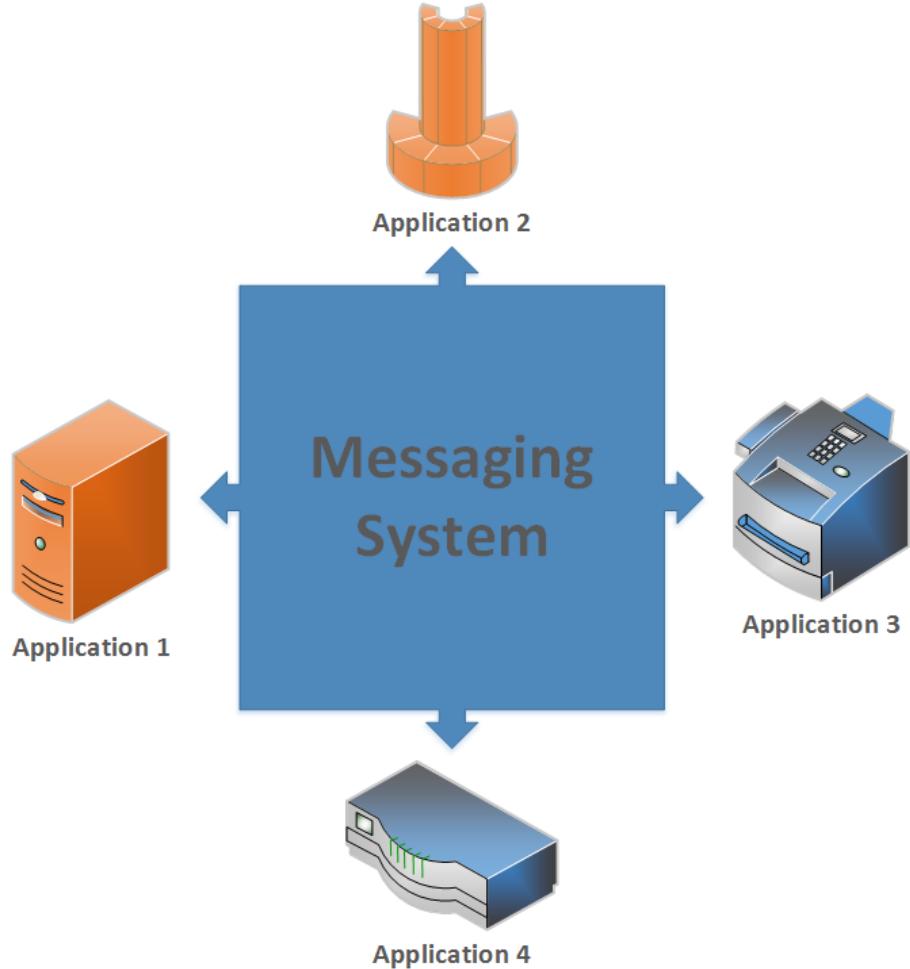
Understanding the principles of messaging systems

In any application integration system design, there are a few important principles that should be kept in mind, such as

- Loose coupling
- Common interface definitions
- Latency
- Reliability.

Understanding messaging systems

- As mentioned earlier, application integration is key for any enterprise to achieve a comprehensive set of functionalities spanning multiple discrete applications.
- To achieve this, applications need to share information in a timely manner.
- A messaging system is one of the most commonly used mechanisms for information exchange in applications.



Understanding messaging systems

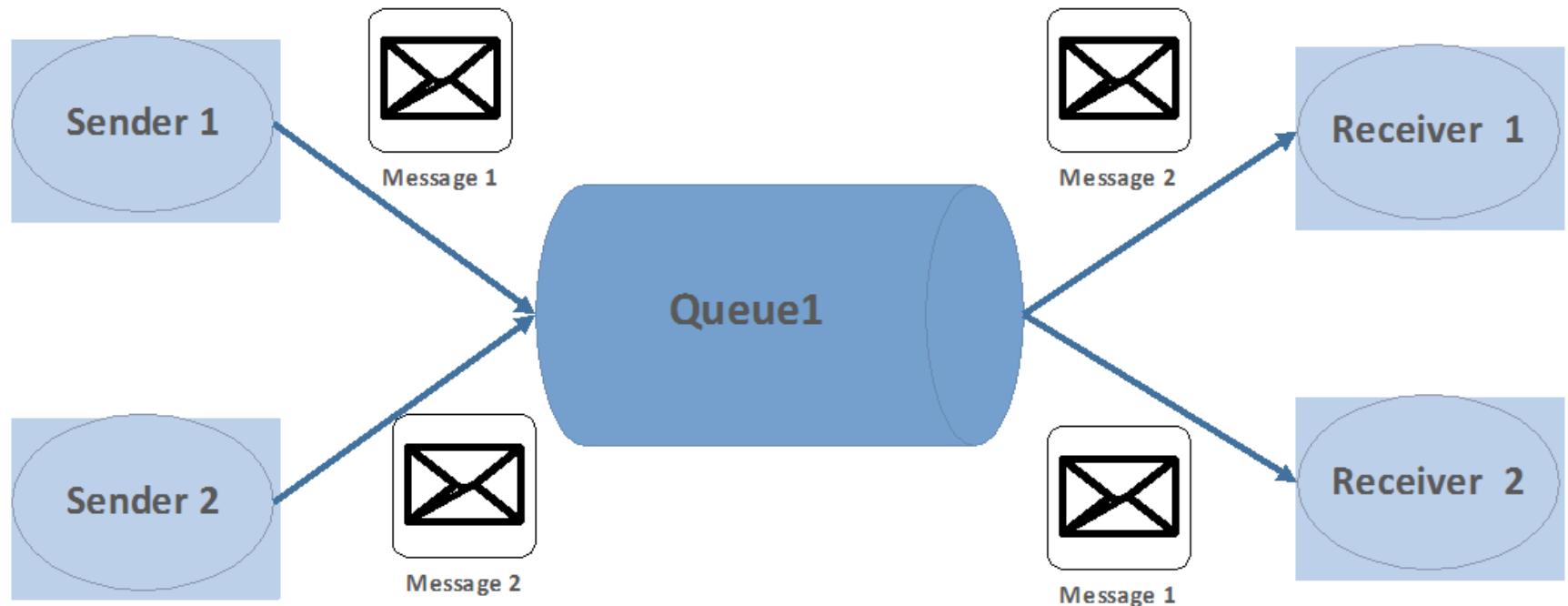
For a start, you should understand some of the basic concepts of any messaging system. Understanding these concepts is beneficial to you as it will help you understand different messaging technologies such as Kafka. The following are some of the basic messaging concepts:

- Message queues
- Messages (data packets)
- Sender (producer)
- Receiver (consumer)
- Data transmission protocols
- Transfer mode:

Peeking into a point-to-point messaging system

- This section focuses on the point-to-point (PTP) messaging model.
- In a PTP messaging model, message producers are called senders and consumers are called receivers.
- They exchange messages by means of a destination called a queue.
- Senders produce messages to a queue and receivers consume messages from this queue.

Peeking into a point-to-point messaging system



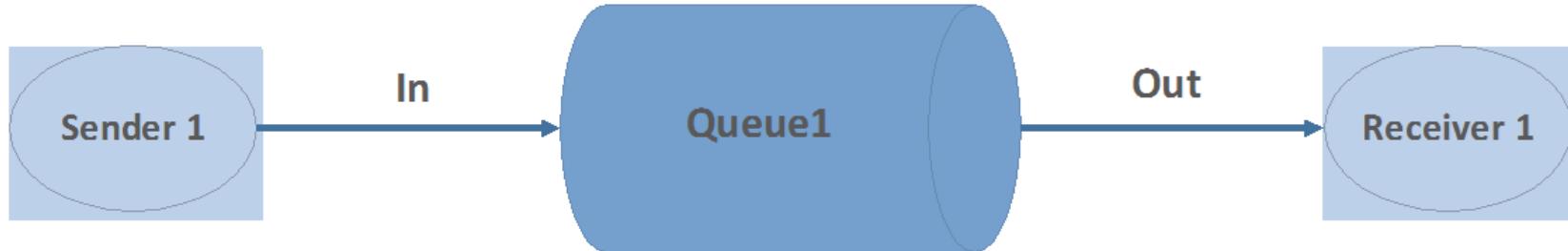
Peeking into a point-to-point messaging system

The PTP messaging model can be further categorized into two types:

- Fire-and-forget model
- Request/reply model

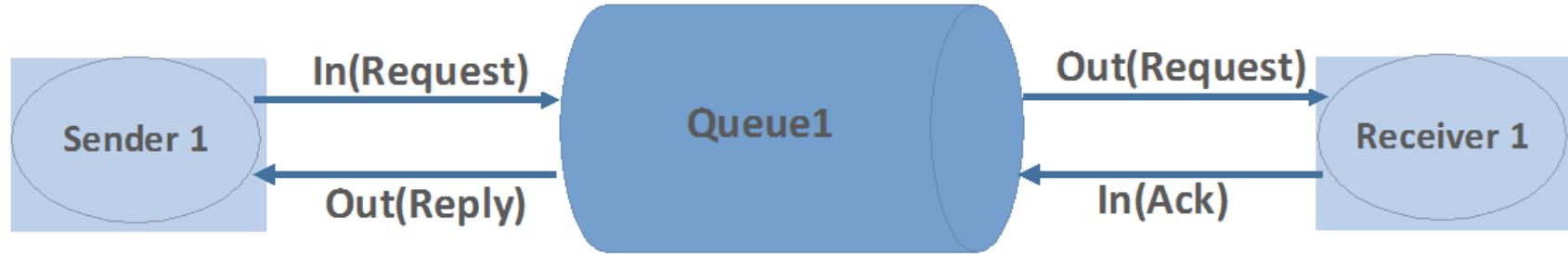
Peeking into a point-to-point messaging system

- For example, you may want to use this method to send a message to a logging system, to alert a system to generate a report, or trigger an action to some other system.
- The following figure presents a fire-and-forget PTP messaging model:



Peeking into a point-to-point messaging system

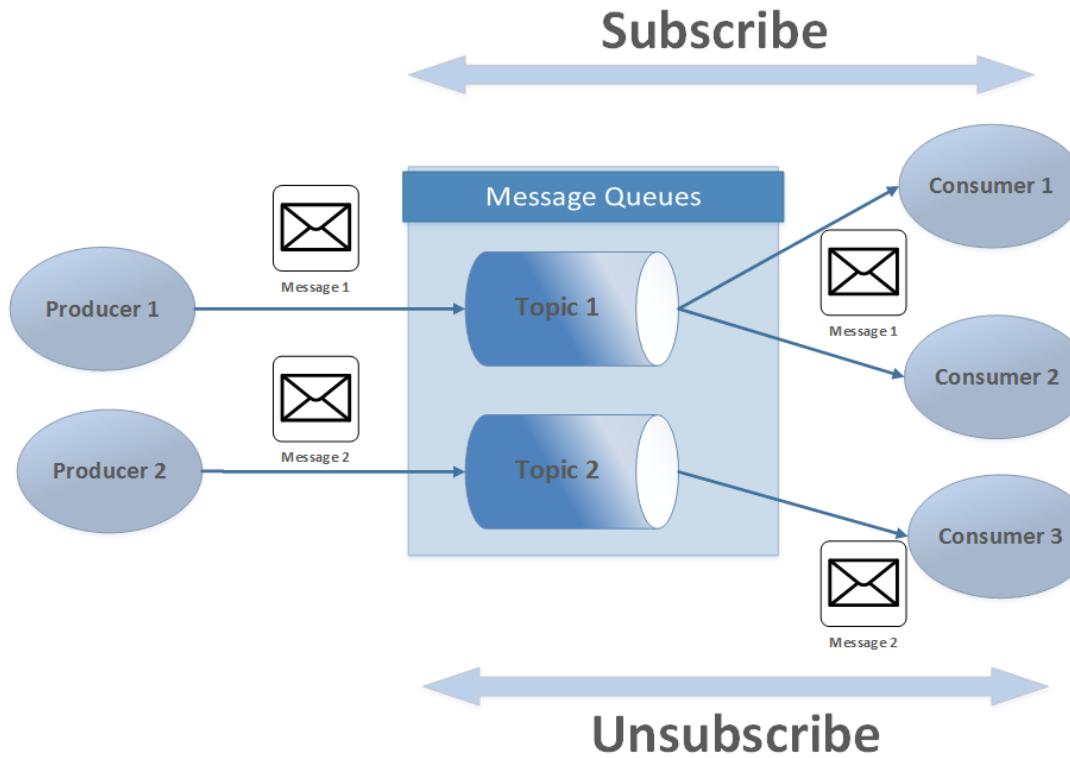
- The following figure represents a request/reply PTP messaging model:

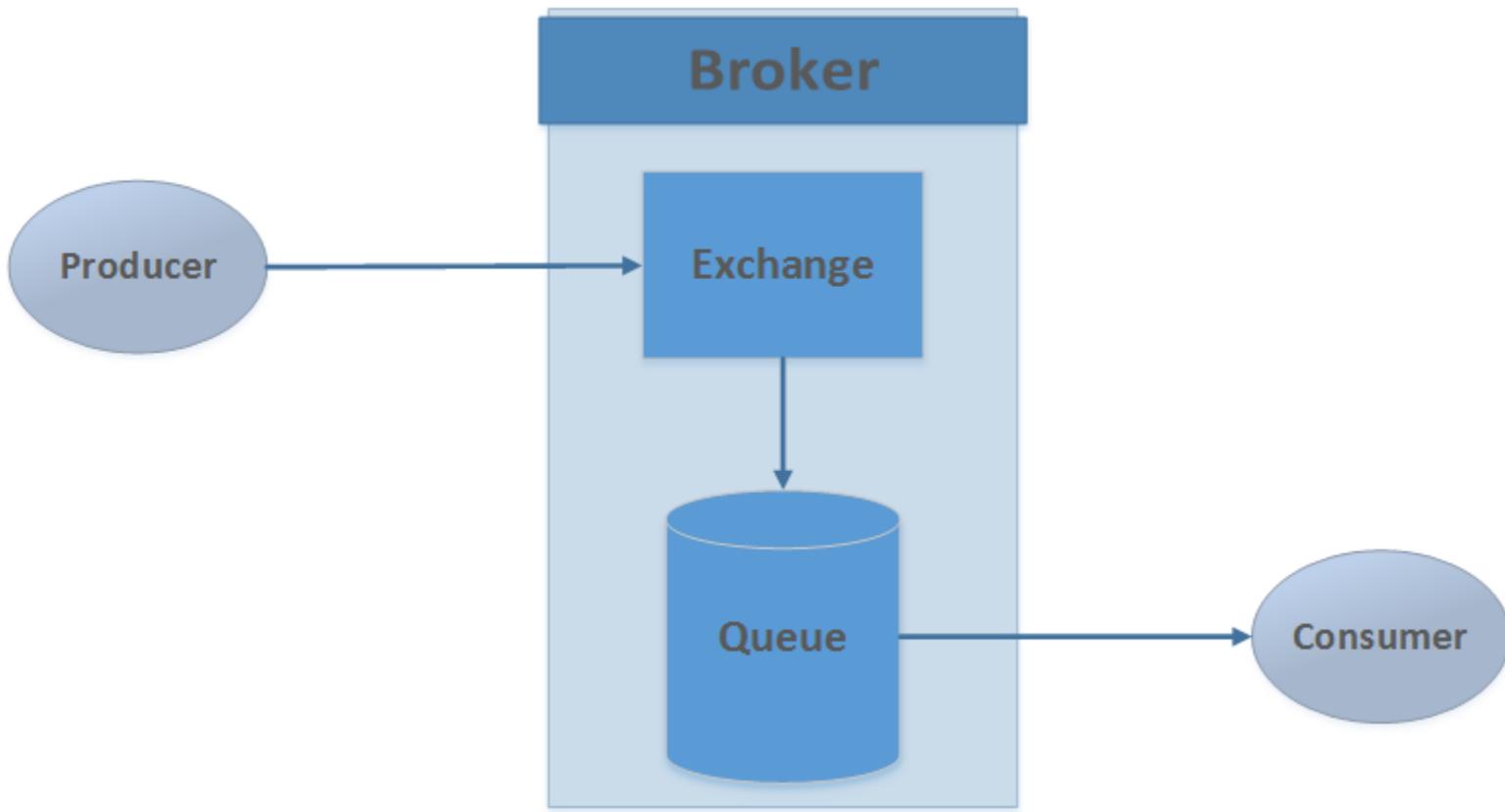


Publish-subscribe messaging system

- A Pub/Sub messaging model is used when you need to broadcast an event or message to many message consumers.
- Unlike the PTP messaging model, all message consumers (called subscribers) listening on the topic will receive the message.

- Consumers can unsubscribe to a queue whenever they want to:





Advance Queuing Messaging Protocol

An AQMP messaging system consists of three main components:

- Publisher(s)
- Consumer(s)
- Broker/server(s)

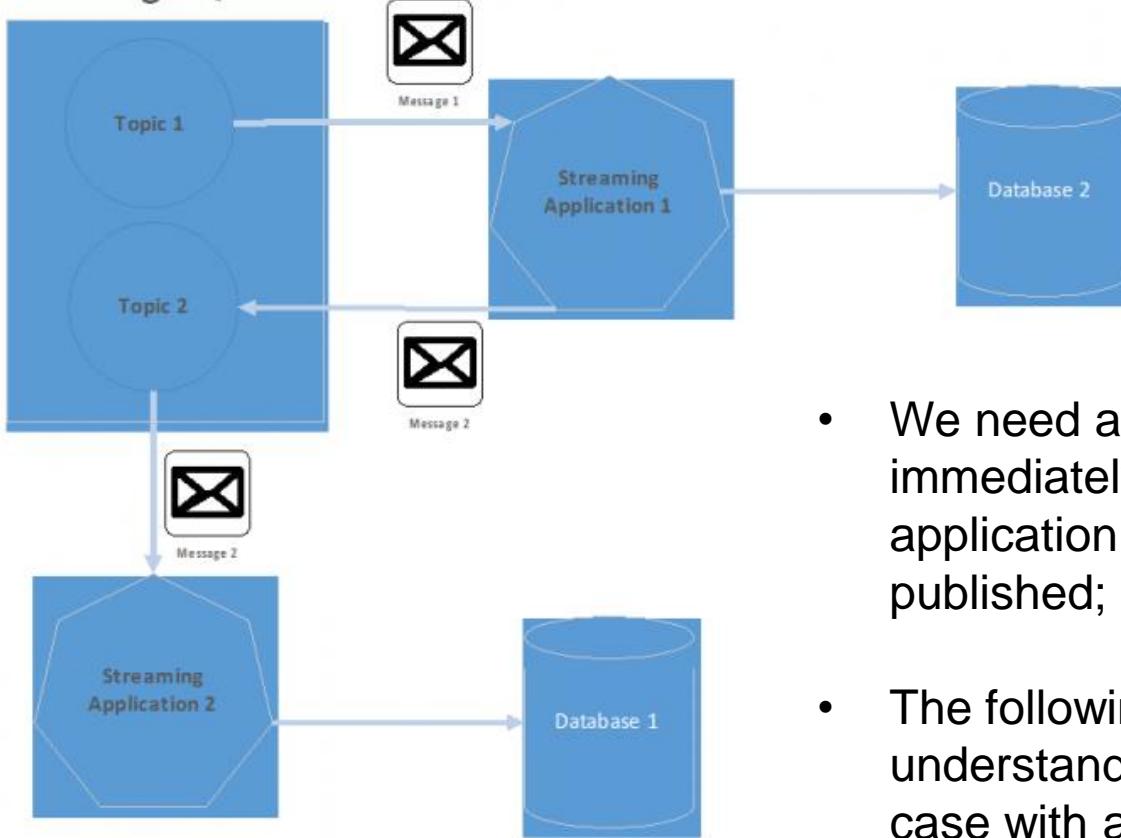
Messaging systems in big data streaming applications

In this section, we will talk about how messaging systems play important role in a big data application.

Let's understand the different layers in a big data application:

- Ingestion layer
- Processing layer
- Consumption layer

Message Queue



- We need a messaging system that immediately tells the streaming application that, Something got published; please process it.
- The following diagram helps you understand a messaging system use case with a streaming application:

Messaging systems in big data streaming applications

Concluding our discussion on messaging systems, these are the points that are important for any streaming application:

- High consuming rate
- Guaranteed delivery
- Persisting capability
- Security
- Fault tolerance

Summary

- In this lesson, we covered concepts of messaging systems. We learned the need for Messaging Systems in Enterprises.
- We further emphasized different ways of using messaging systems such as point to point or publish/subscribe.
- We introduced Advance Message Queuing Protocol (AQMP) as well.

No Lab

2. Introducing Kafka, the Distributed Messaging Platform

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or professional environment.

Introducing Kafka, the Distributed Messaging Platform

We will cover the following topics in this lesson:

- Kafka origins
- Kafka's architecture
- Message topics
- Message partitions
- Replication and replicated logs
- Message producers
- Message consumers
- Role of Zookeeper

Kafka origins

- Most of you must have used the LinkedIn portal in your professional career.
- The Kafka system was first built by the LinkedIn technical team.
- LinkedIn constructed a software metrics collecting system using custom in-house components with some support from existing open-source tools.
- The system was used to collect user activity data on their portal.

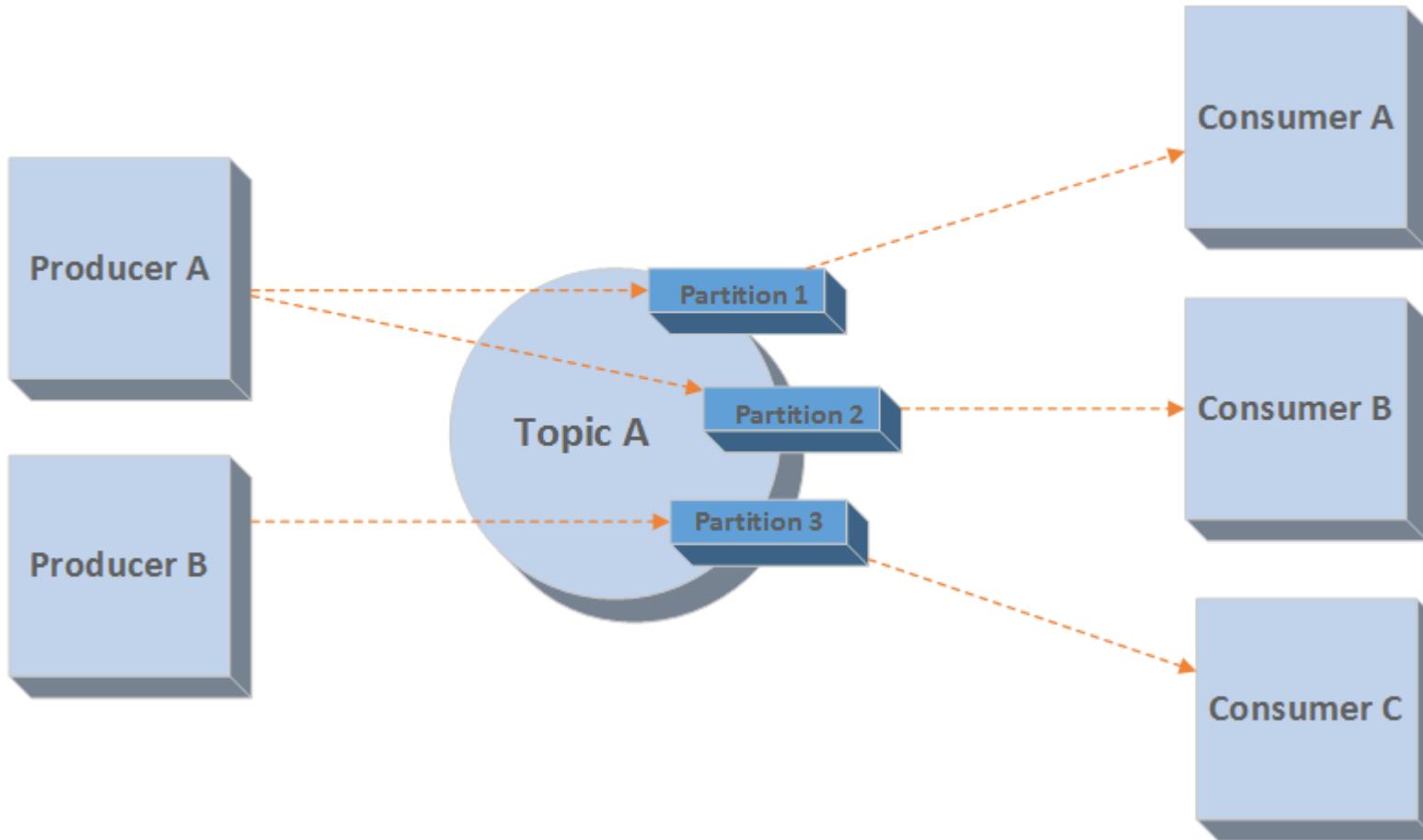
Kafka origins

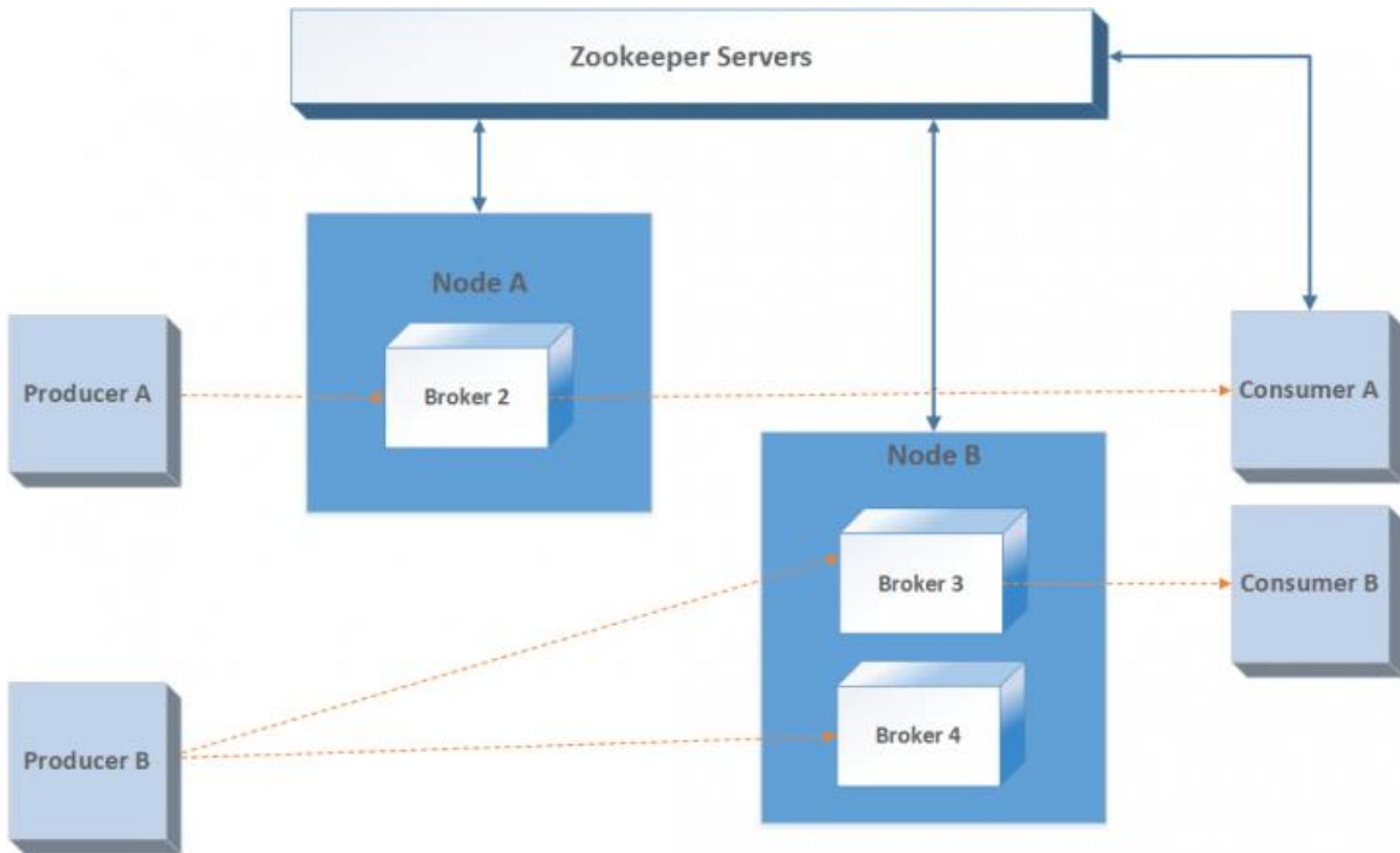
Kafka was built with the following goals in mind:

- Loose coupling between message Producers and message Consumers
- Persistence of message data to support a variety of data consumption scenarios and failure handling
- Maximum end-to-end throughput with low latency components
- Managing diverse data formats and types using binary data formats
- Scaling servers linearly without affecting the existing cluster setup

Kafka's architecture

- Every message in Kafka topics is a collection of bytes, This collection is represented as an array. Producers are the applications that store information in Kafka queues.
- They send messages to Kafka topics that can store all types of messages, Every topic is further differentiated into partitions.
- Each partition stores messages in the sequence in which they arrive.





Kafka's architecture

- A typical Kafka cluster consists of multiple brokers.
- It helps in load-balancing message reads and writes to the cluster.
- Each of these brokers is stateless, However, they use Zookeeper to maintain their states.
- Each topic partition has one of the brokers as a leader and zero or more brokers as followers, The leaders manage any read or write requests for their respective partitions

Kafka's architecture

- Zookeeper is an important component of a Kafka cluster & It manages and coordinates Kafka brokers and consumers.
- Zookeeper keeps track of any new broker additions or any existing broker failures in the Kafka cluster.
- Accordingly, it will notify the producer or consumers of Kafka queues about the cluster state. This helps both producers and consumers in coordinating work with active brokers

Message topics

- If you are into software development and services, I am sure you will have heard terms such as database, tables, records, and so on.
- In a database, we have multiple tables; let's say, Items, Price, Sales, Inventory, Purchase, and many more, Each table contains data of a specific category.
- There will be two parts in the application: one will be inserting records into these tables and the other will be reading records from these tables.

Message partitions

- Suppose that we have in our possession a purchase table
- We want to read records for an item from the purchase table that belongs to a certain category, say, electronics. In the normal course of events
- We will simply filter out other records, but what if we partition our table in such a way that we will be able to read the records of our choice quickly?

Message partitions

Let's understand the pros and cons of a large number of partitions:

- High throughput: Partitions are a way to achieve parallelism in Kafka.
- Write operations on different partitions happen in parallel.
- All time-consuming operations will happen in parallel as well; this operation will utilize hardware resources at the maximum.

Message partitions

Throughput for the producer and consumer can be increased or decreased based on different configurations that we will discuss in detail in upcoming lessons.

- Increases producer memory: You must be wondering how increasing the number of partitions will force us to increase producer memory.
- A producer does some internal stuff before flushing data to the broker and asking them to store it in the partition.

Message partitions

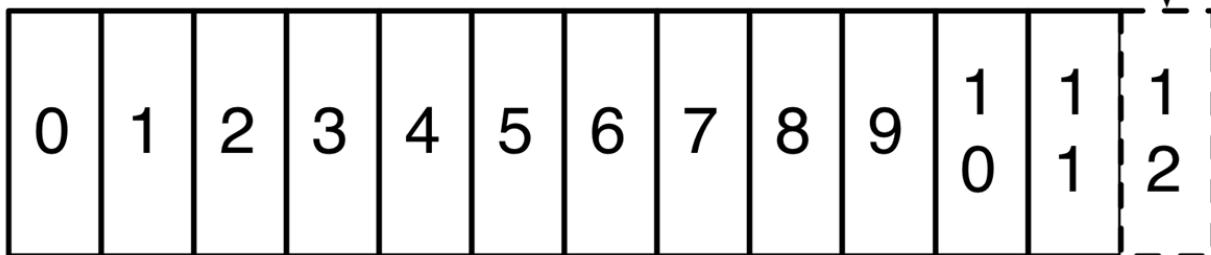
- High availability issue: Kafka is known as high-availability, high-throughput, and distributed messaging system.

The general formula is:

Delay Time = (Number of Partition/replication * Time to read metadata for single partition)

Producers

writes



reads

Consumer A
(offset=9)

Consumer B
(offset=11)

Replication and replicated logs

- Replication is one of the most important factors in achieving reliability for Kafka systems.
- Replicas of message logs for each topic partition are maintained across different servers in a Kafka cluster.
- This can be configured for each topic separately.
What it essentially means is that for one topic, you can have the replication factor as 3 and for another, you can use 5.

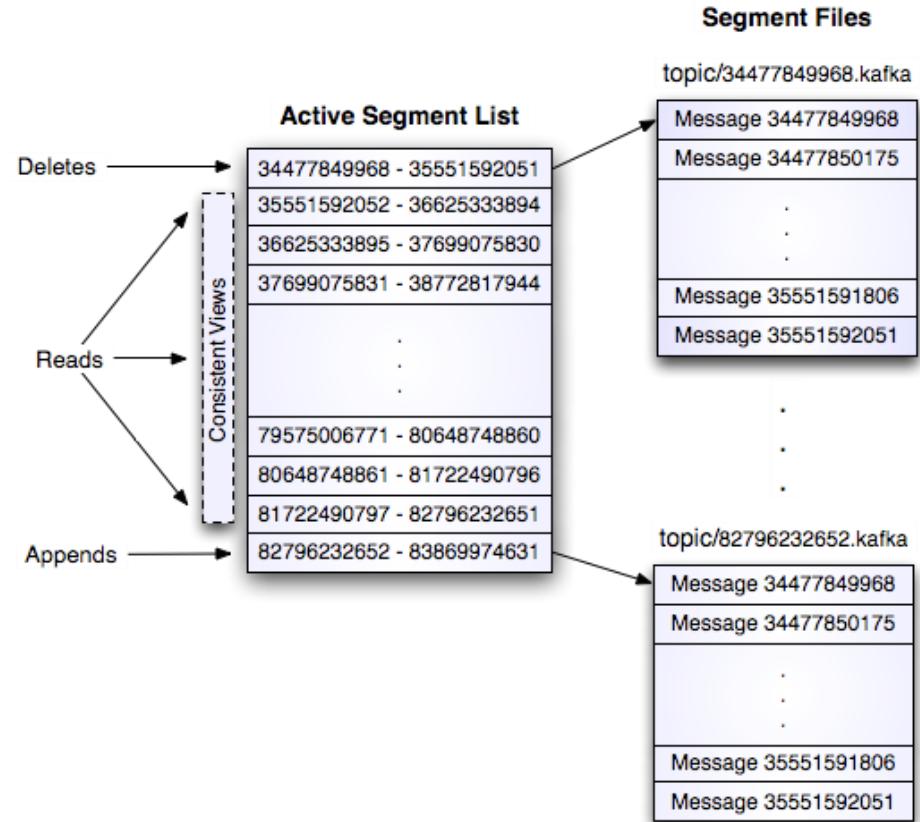
Replication and replicated logs

To maintain such replica consistency, there are two approaches. In both approaches, there will be a leader through which all the read and write requests will be processed. There is a slight difference in replica management and leader election:

- Quorum-based approach
- Primary backup approach

Kafka Log Implementation

- Here is the diagram that will clear the Kafka Log Implementation:



Message producers

- In Kafka, the producer is responsible for sending data to the partition of the topic for which it is producing data.
- The hash value is generally calculated by the message key that we provide when writing the message to a Kafka topic.

Message consumers

- The consumer is anyone who subscribes for topics in Kafka.
- Each consumer belongs to a consumer group and some consumer groups contains multiple consumers. Consumers are an interesting part of Kafka and we will cover them in detail.

Role of Zookeeper

We are dedicating a separate section to the role of Zookeeper in the Kafka cluster. Kafka cannot work without Zookeeper. Kafka uses Zookeeper for the following functions:

- Choosing a controller
- Brokers metadata
- Topic metadata
- Client quota information
- Kafka topic ACLs

Summary

- We have come to the end of this lesson, and by now you should have a basic understanding of the Kafka messaging system.
- An important aspect of mastering any system is that you should understand the system end to end at a high level first.
- This will put you in a better position when you understand individual components of the system in detail.

Complete lab 2

3. Deep Dive into Kafka Producers



Deep Dive into Kafka Producers

We will cover the following topics in this lesson:

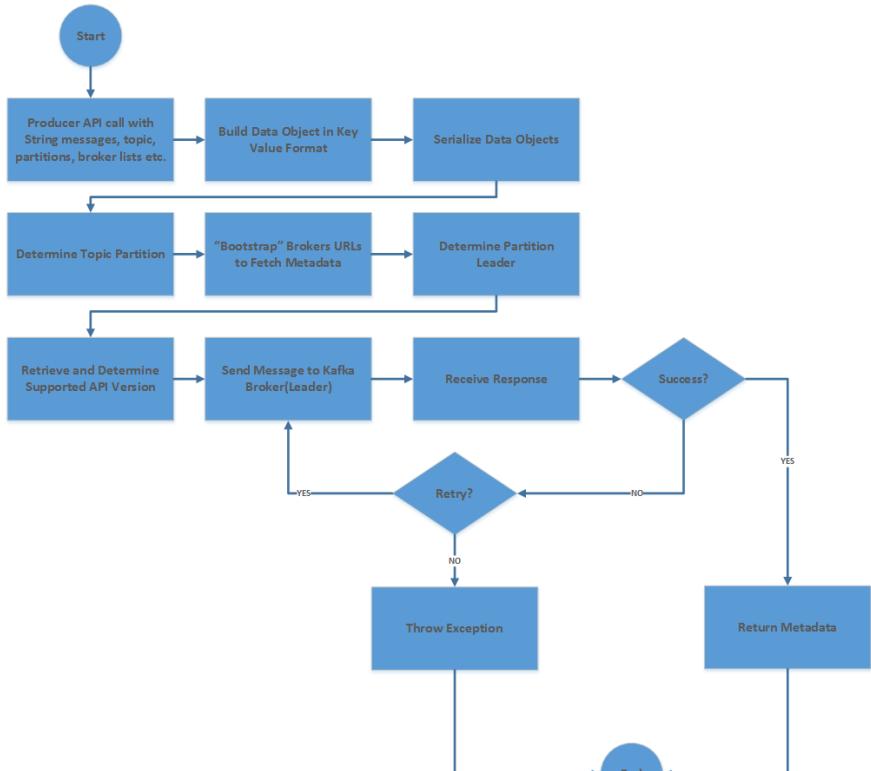
- Internals of a Kafka producer
- The Kafka Producer API and its uses
- Partitions and their uses
- Additional configuration for producers
- Some common producer patterns
- An example of a producer
- Best practices to be followed for a Kafka producer

Kafka producer internals

First, we need to understand the responsibilities of Kafka producers apart from publishing messages. Let's look at them:

- Bootstrapping Kafka broker URLs
- Data serialization
- Determining topic partition
- Determining the leader of the partition
- Failure handling/retry ability
- Batching

- The following image shows the high-level steps involved in producing messages to the Kafka cluster:



Batching/One At a Time

Kafka Producer APIs

Creating a Kafka producer involves the following steps:

- Required configuration.
- Creating a producer object.
- Setting up a producer record.
- Creating a custom partition if required.
- Additional configuration.

Kafka Producer APIs

- Here is how Java works for Producer APIs:

```
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "broker1:port,broker2:port");
producerProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
KafkaProducer<String, String> producer = new
KafkaProducer<String, String>(producerProps);
```

Kafka Producer APIs

- The Producer API in Scala:

```
val producerProps = new Properties()  
producerProps.put("bootstrap.servers",  
"broker1:port,broker2:port");
```

```
    producerProps.put("key.serializer",  
"org.apache.kafka.common.serialization.StringSerializer")  
    producerProps.put("value.serializer",  
"org.apache.kafka.common.serialization.StringSerializer")
```

```
val producer = new KafkaProducer[String, String](producerProps)
```

Producer object and ProducerRecord object

Partition number, timestamp, and key are optional parameters, but the topic to which data will be sent and value that contains the data is mandatory.

- If the partition number is specified, then the specified partition will be used when sending the record
- If the partition is not specified but a key is specified, a partition will be chosen using a hash of the key
- If both key and partition are not specified, a partition will be assigned in a round-robin fashion

Producer object and ProducerRecord object

- Here is the producerRecord in Java:

```
ProducerRecord producerRecord = new ProducerRecord<String,  
String>(topicName, data);  
Future<RecordMetadata> recordMetadata =  
producer.send(producerRecord);
```

- Here is an example of producerRecord in Scala:

```
val producerRecord = new ProducerRecord<String,  
String>(topicName, data);  
val recordMetadata = producer.send(producerRecord);
```

Producer object and ProducerRecord object

- Here is the first constructor for producerRecord:
`ProducerRecord(String topicName, Integer
numberOfpartition, K key, V value)`
- The second constructor goes something like this:
`ProducerRecord(String topicName, Integer
numberOfpartition, Long timestamp, K key, V value)`

Producer object and ProducerRecord object

- The third constructor is as follows:

ProducerRecord(String topicName, K key, V value)

- The final constructor of our discussion is as follows:

ProducerRecord(String topicName, V value)

Producer object and ProducerRecord object

- Java:

```
ProducerRecord Recorded = new ProducerRecord<String,  
String>(topicName, data);  
Object recordMetadata = producer.send(producerRecord).get();
```

- Scala:

```
val producerRecord = new ProducerRecord<String,  
String>(topicName, data);  
val recordMetadata = producer.send(producerRecord);
```

Producer object and ProducerRecord object

- Asynchronous messaging: Sometimes, we have a scenario where we do not want to deal with responses immediately or we do not care about losing a few messages and we want to deal with it after some time.
- Kafka provides us with the callback interface that helps in dealing with message reply, irrespective of error or successful. `send()` can accept an object that implements the callback interface.

`send(ProducerRecord<K,V> record,Callbackcallback)`

Producer object and ProducerRecord object

- Here is the example in Java:

```
public class ProducerCallback implements Callback {  
    public void onCompletion(RecordMetadata Producer Callback,  
    Exception ex) {  
        if(ex!=null){  
            //deal with exception here  
        }  
        else{  
            //deal with RecordMetadata here  
        }  
    }  
}
```

Producer object and ProducerRecord object

- Scala:

```
class ProducerCallback extends Callback {  
    override def onCompletion(recordMetadata: RecordMetadata, ex:  
        Exception): Unit = {  
        if (ex != null) {  
            //deal with exception here  
        }  
        else {  
            //deal with RecordMetadata here  
        }  
    }  
}
```

Producer object and ProducerRecord object

- Once we have the Callback class implemented, we can simply use it in the send method as follows:

```
val callBackObject = producer.send(producerRecord,new  
ProducerCallback());
```

- If Kafka has thrown an exception for the message, we will not have a null exception object.

Custom partition

- Remember that we talked about key serializer and value serializer as well as partitions used in Kafka producer.
- As of now, we have just used the default partitioner and inbuilt serializer.
- Let's see how we can create a custom partitioner.

Producer object and ProducerRecord object

- Here is an example in Java:
Refer to the file 3_1.txt
- Here is an example in Java:
Refer to the file 3_2.txt

Additional producer configuration

There are other optional configuration properties available for Kafka producer that can play an important role in performance, memory, reliability, and so on:

- `buffer.memory`: This is the amount of memory that producer can use to buffer a message that is waiting to be sent to the Kafka server.
- In simple terms, it is the total memory that is available to the Java producer to collect unsent messages.

Java Kafka producer example

Prerequisite

- IDE: We recommend that you use a Scala-supported IDE such as IDEA, NetBeans, or Eclipse. We have used JetBrains
IDEA:[https://www.jetbrains.com/idea/download/.](https://www.jetbrains.com/idea/download/)
- Build tool: Maven, Gradle, or others. We have used Maven to build our project.

Java Kafka producer example

- Pom.xml: Add Kafka dependency to the pom file:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>0.10.0.0</version>
</dependency>
```

Java Kafka producer example

- Java:

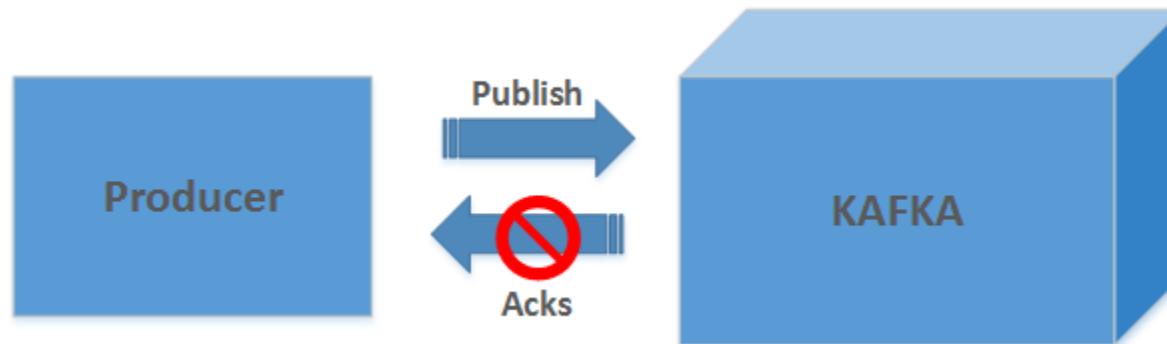
Refer to the file 3_3.txt

Scala:

Refer to the file 3_4.txt

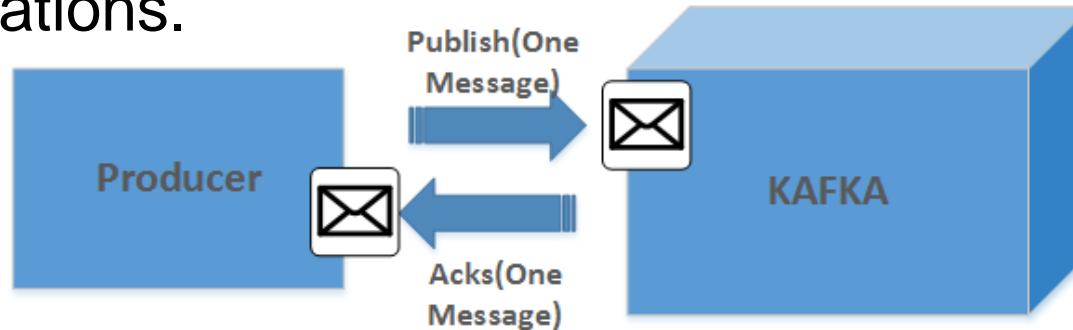
Common messaging publishing patterns

- To use the fire and forget model with Kafka, you have to set producer acks config to 0.
- The following image represents the Kafka-based fire and forget model:



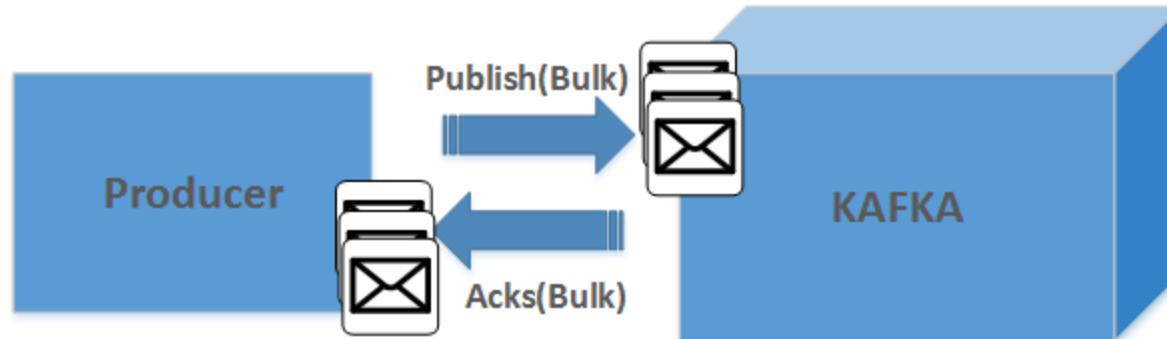
Common messaging publishing patterns

- In this model, producer thread waits for response from Kafka.
- However, this does not mean that you cannot send multiple messages at a time.
- You can achieve that using multithreaded producer applications.



Common messaging publishing patterns

- All the data in one batch would be written in one sequential fashion on hard drives.
- The following image indicates the batching message model:



Common messaging publishing patterns

Let's go through some of the most common best practices to design a good producer application:

- Data validation
- Exception handling
- Number of retries
- Number of bootstrap URLs
- Avoid poor partitioning mechanism
- Temporary persistence of messages
- Avoid adding new partitions to existing topics

Summary

- This concludes our section on Kafka producers, This lesson addresses one of the key functionalities of Kafka message flows.
- The major emphasis in this lesson was for you to understand how Kafka producers work at the logical level and how messages are passed from Kafka producers to Kafka queues.

Complete lab 3

4. Deep Dive into Kafka Consumers



Deep Dive into Kafka Consumers

We will cover the following topics in this lesson:

- Kafka consumer internals
- Kafka consumer APIs
- Java Kafka consumer example
- Scala Kafka consumer example
- Common message consuming patterns
- Best practices

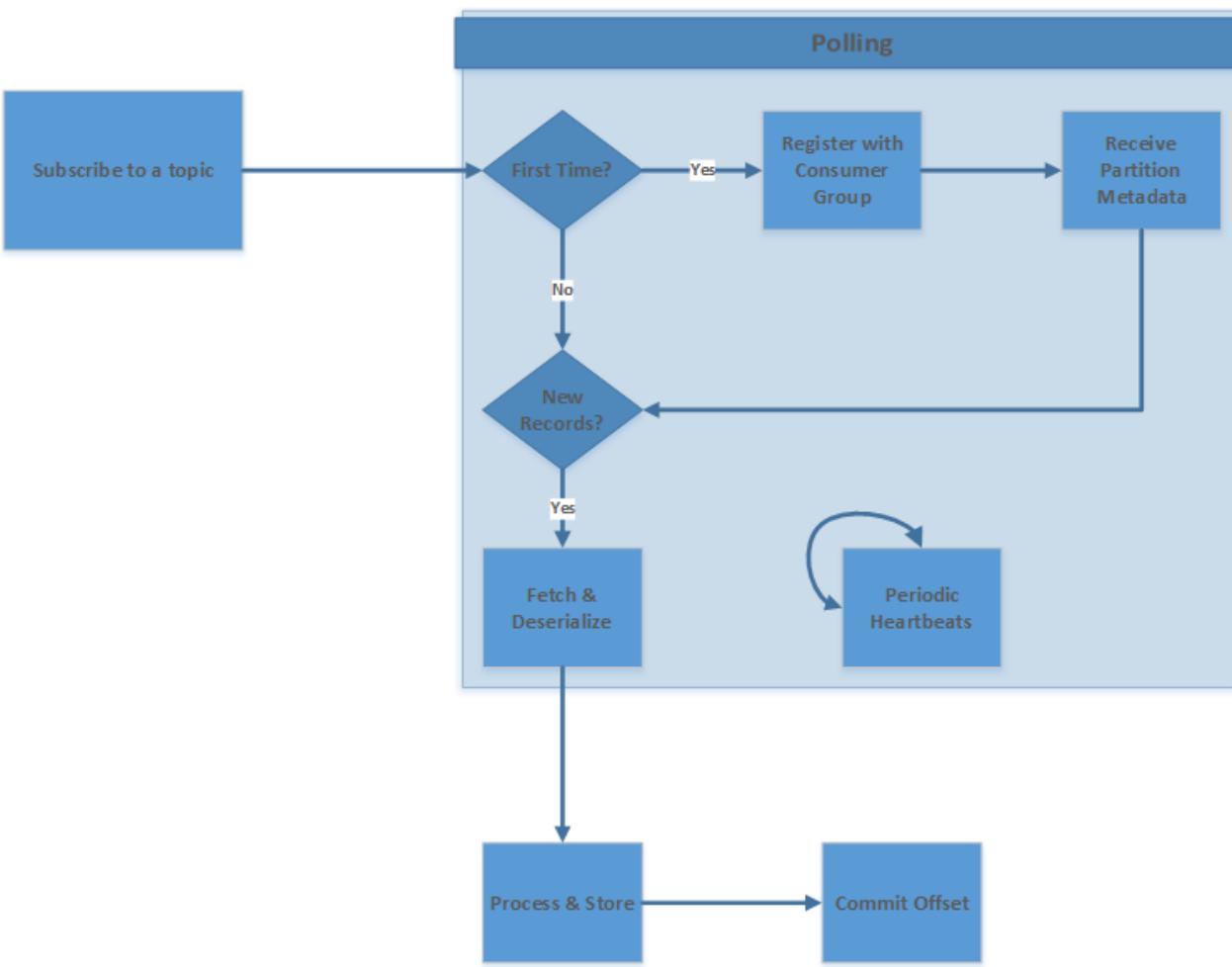
Kafka consumer internals

- Irrespective of the fact that you do not have to code for most of the consumer internal work, you should understand these internal workings thoroughly.
- These concepts will definitely help you in debugging consumer applications and also in making the right application decision choices.

Understanding the responsibilities of Kafka consumers

On the same lines of the previous lesson on Kafka producers, we will start by understanding the responsibilities of Kafka consumers apart from consuming messages from Kafka queues.

- Subscribing to a topic
- Consumer offset position
- Replay / rewind / skip messages
- Heartbeats
- Offset commits
- Deserialization



Kafka consumer APIs

We will see how this concept helps in building a good consumer application.

- Consumer configuration
- KafkaConsumer object
- Subscription and polling
- Commit and offset
- Additional configuration

Consumer configuration

- Let's see how we set and create this in the real programming world.
- Java:

```
Properties consumerProperties = new Properties();
consumerProperties.put("bootstrap.servers", "10.200.99.197:6667");
consumerProperties.put("group.id", "Demo");
consumerProperties.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
consumerProperties.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(consumerProperties);
```

Consumer configuration

- Scala:

```
val consumerProperties: Properties = new Properties();
consumerProperties.put("bootstrap.servers", "10.200.99.197:6667")
consumerProperties.put("group.id", "consumerGroup1")
consumerProperties.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer")
consumerProperties.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer")
val consumer: KafkaConsumer[String, String] = new
KafkaConsumer[String, String](consumerProperties)
```

Subscription and polling

- Consumer has to subscribe to some topic to receive data.
- The KafkaConsumer object has subscribe(), which takes a list of topics that the consumer wants to subscribe to.
- There are different forms of the subscribe method.

Committing and polling

- Polling is fetching data from the Kafka topic. Kafka returns the messages that have not yet been read by consumer.
- How does Kafka know that consumer hasn't read the messages yet?
- Consumer needs to tell Kafka that it needs data from a particular offset and therefore, consumer needs to store the latest read message somewhere so that in case of consumer failure, consumer can start reading from the next offset.

Committing and polling

- Java:

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(2);  
    for (ConsumerRecord<String, String> record : records)  
        System.out.printf("offset = %d, key = %s, value = %sn",  
                           record.offset(), record.key(), record.value());  
    try {  
        consumer.commitSync();  
    } catch (CommitFailedException ex) {  
        //Logger or code to handle failed commit  
    }  
}
```

- Scala:

```
while (true) {
    val records: ConsumerRecords[String, String] = consumer.poll(2)
    import scala.collection.JavaConversions._
    for (record <- records) println("offset = %d, key = %s, value = %sn",
        record.offset, record.key, record.value)

    try
        consumer.commitSync()

    catch {
        case ex: CommitFailedException =>
            //Logger or code to handle failed commit
    }
}
```

Asynchronous commit:

- Java:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(2);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %sn",
                           record.offset(), record.key(), record.value());
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
OffsetAndMetadata> map, Exception e) {
            }
        });
    }
}
```

Asynchronous commit:

- Scala:

```
while (true) {  
    val records: ConsumerRecords[String, String] = consumer.poll(2)  
    for (record <- records) println("offset = %d, key = %s, value = %sn",  
        record.offset, record.key, record.value)  
    consumer.commitAsync(new OffsetCommitCallback {  
        override def onComplete(map: util.Map[TopicPartition,  
            OffsetAndMetadata], ex: Exception): Unit = {  
            }  
        }  
    })  
}
```

Java Kafka consumer

- The following program is a simple Java consumer which consumes data from topic test.
- Please make sure data is already available in the mentioned topic otherwise no record will be consumed.

Refer to the file 4_1.txt

Scala Kafka consumer

- This is the Scala version of the previous program and will work the same as the previous snippet.
- Kafka allows you to write consumer in many languages including Scala.

Refer to the file 4_2.txt

Rebalance listeners

- Kafka provides you with an API to handle such scenarios.
- It provides the ConsumerRebalanceListener interface that contains the onPartitionsRevoked() and onPartitionsAssigned() methods.
- We can implement these two methods and pass an object while subscribing to the topic using the subscribe method discussed earlier:

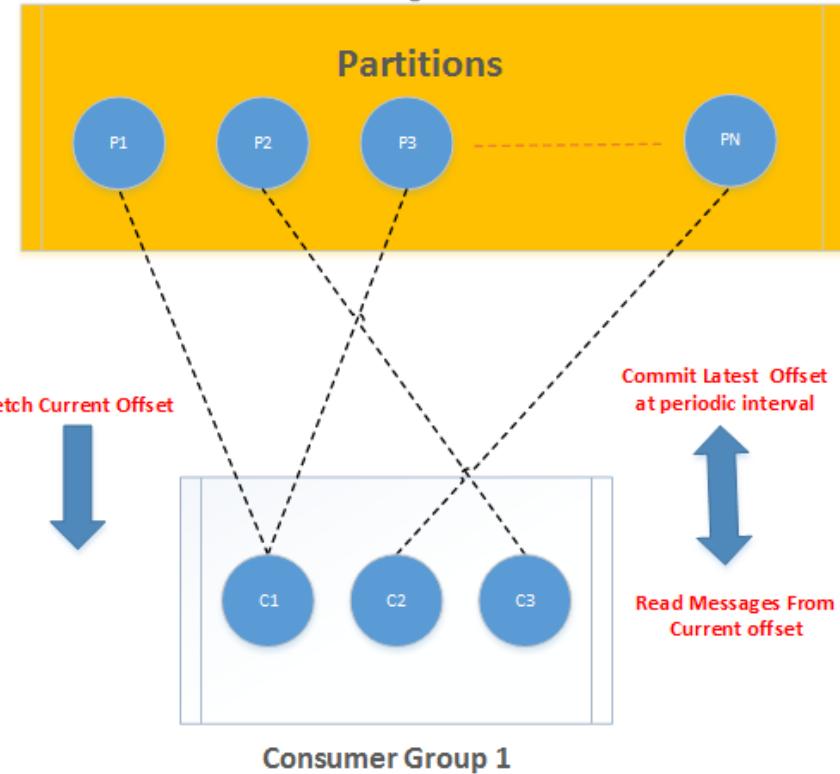
Refer to the file 4_3.txt

Common message consuming patterns

Here are a few of the common message consuming patterns:

- Consumer group - continuous data processing: In this pattern, once consumer is created and subscribes to a topic, it starts receiving messages from the current offset.
- The consumer commits the latest offsets based on the count of messages received in a batch at a regular, configured interval.

Topic 1



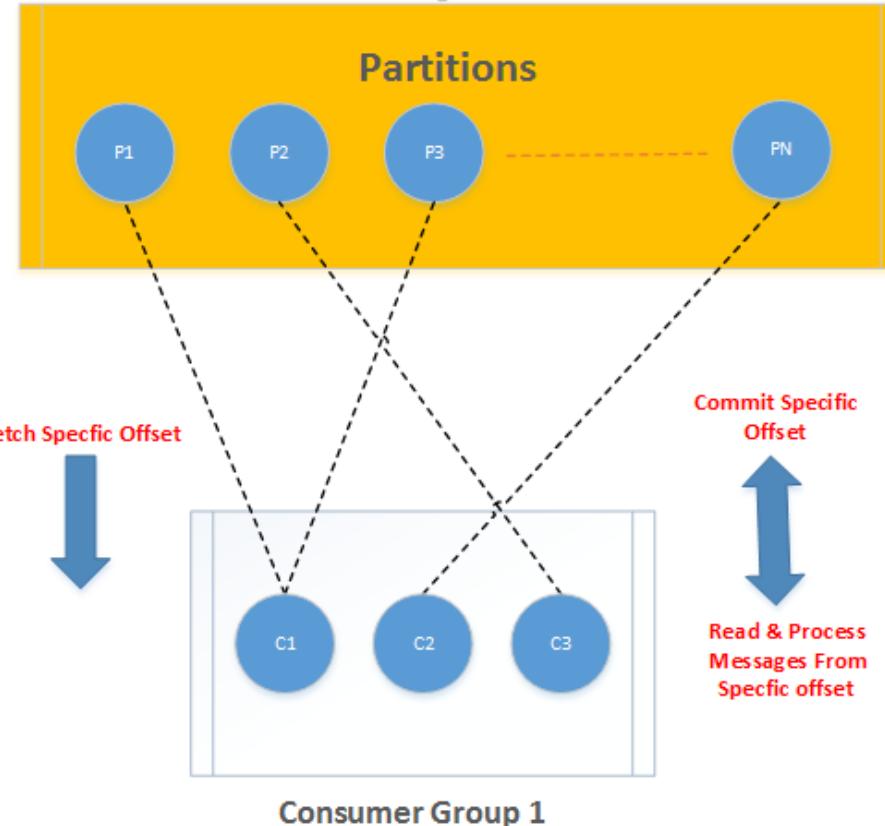
- The following image represents the continuous data processing pattern:

Common message consuming patterns

Consumer group - continuous data processing

- Consumer group - discrete data processing:
Sometimes you want more control over consuming messages from Kafka.
- You want to read specific offsets of messages that may or may not be the latest current offset of the particular partition.
- Subsequently, you may want to commit specific offsets and not the regular latest offsets.

Topic 1



Best practices

After going through the lesson, it is important to note a few of the best practices. They are listed as follows:

- Exception handling
- Handling rebalances
- Commit offsets at the right time
- Automatic offset commits

Summary

- This concludes our section on Kafka consumers, This lesson addresses one of the key functionalities of Kafka message flows.
- The major focus was on understanding consumer internal working and how the number of consumers in the same group and number of topic partitions can be utilized to increase throughput and latency.

Complete lab 4

5. Building Spark Streaming Applications with Kafka

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or programming environment.

Building Spark Streaming Applications with Kafka

In short, we will cover the following topics:

- Introduction to Spark
- Internals of Spark such as RDD
- Spark Streaming
- Receiver-based approach (Spark-Kafka integration)
- Direct approach (Spark-Kafka integration)
- Use case (Log processing)

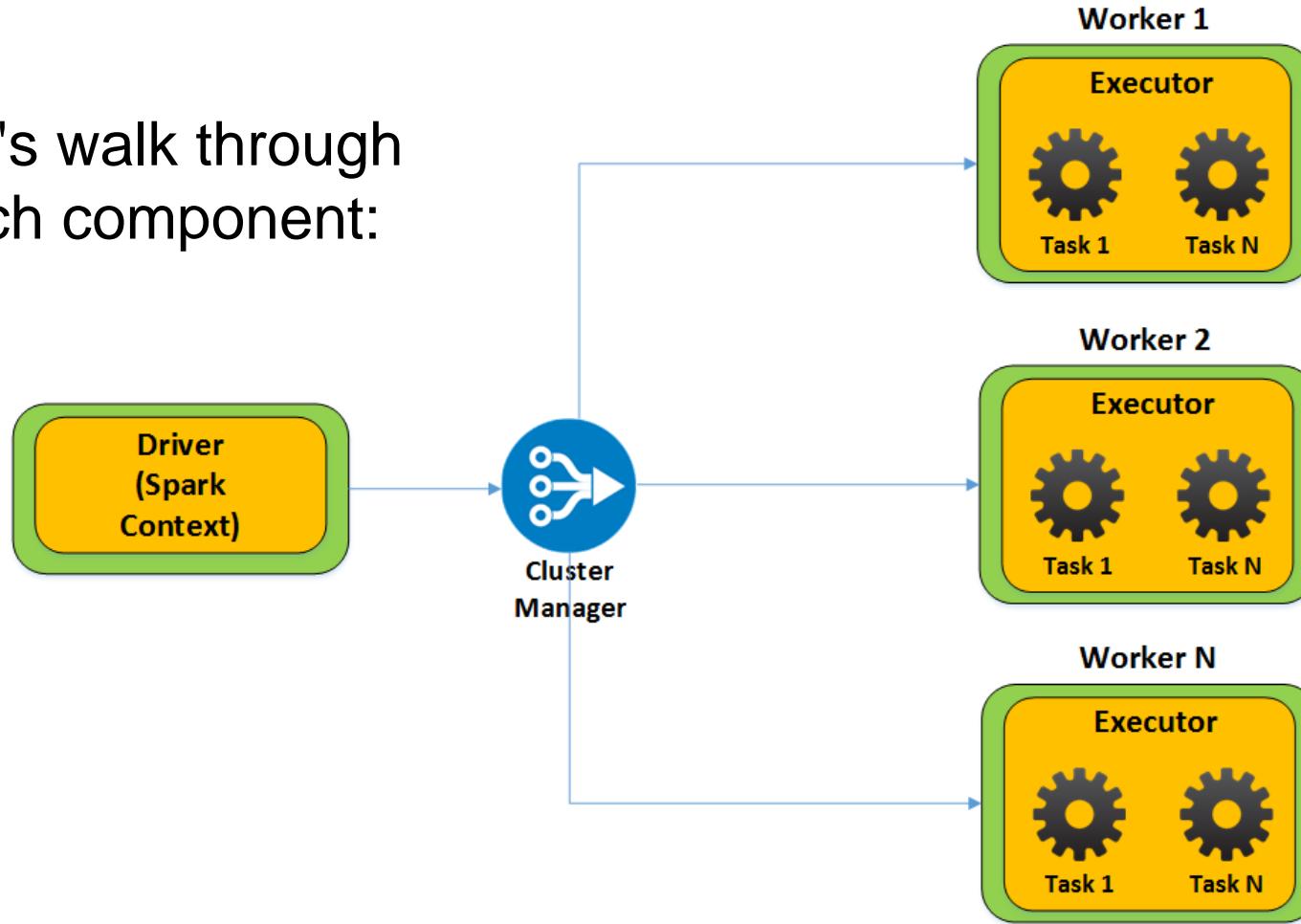
Introduction to Spark

- Apache Spark is distributed in-memory data processing system.
- It provides rich set of API in Java, Scala, and Python.
- Spark API can be used to develop applications which can do batch and real-time data processing and analytics, machine learning, and graph processing of huge volumes of data on a single clustering platform.

Spark architecture

- Like Hadoop, Spark also follows the master/slave architecture, master daemons called Spark drivers, and multiple slave daemons called executors. Spark runs on a cluster and uses cluster resource managers such as YARN, Mesos, or Spark Standalone cluster manager.

- Let's walk through each component:

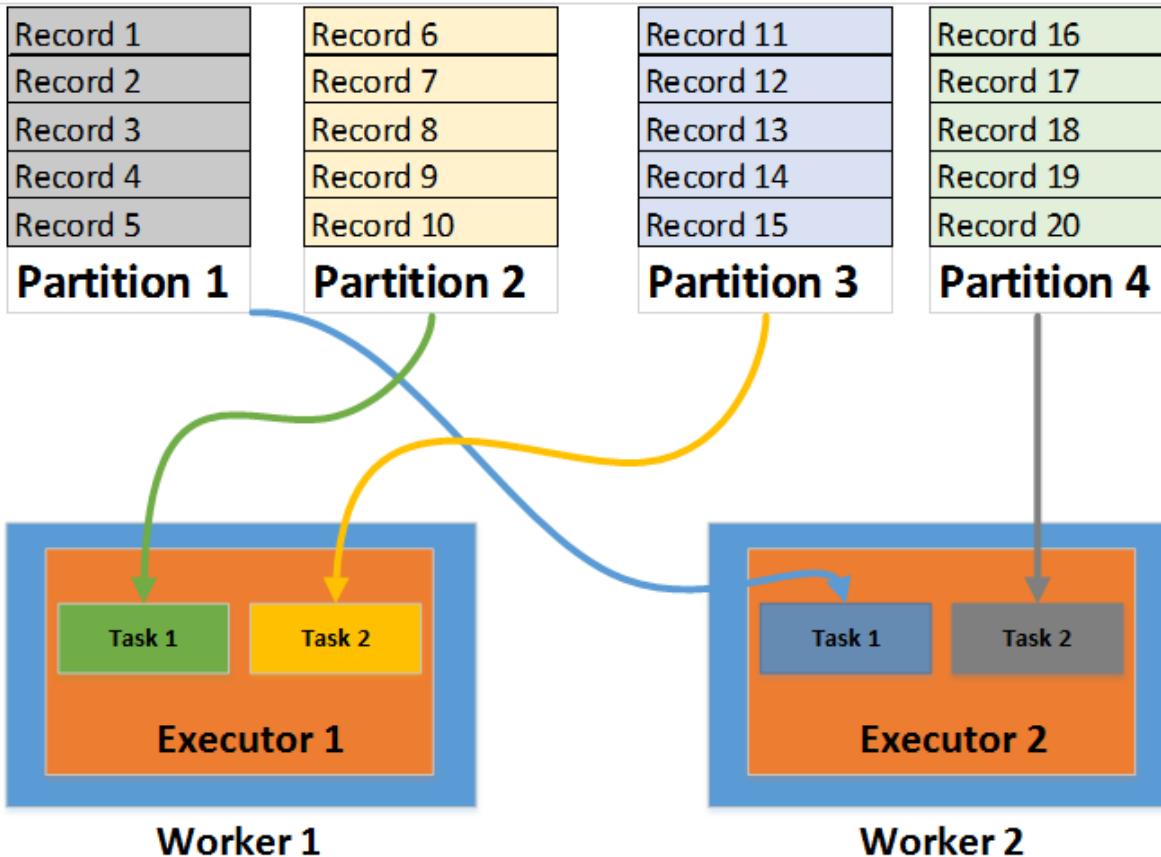


Pillars of Spark

The following are the important pillars of Spark:

- Resilient Distributed Dataset (RDD): RDD is the backbone of Spark. RDD is an immutable, distributed, fault tolerant collection of objects.
- RDDs are divided into logical partitions which are computed on different worker machines.

Record RDD

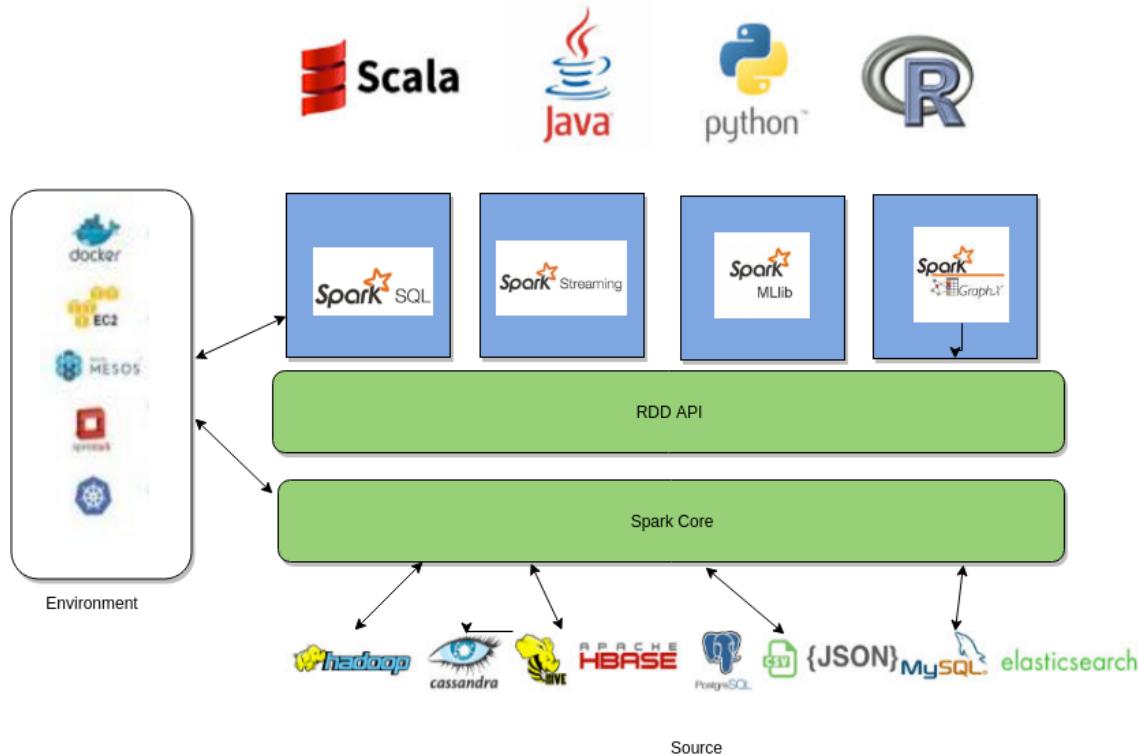


Directed acyclic graph (DAG)

- Directed acyclic graph (DAG): Let's take the following example of a word count in Spark:

```
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
val input = sc.textFile(inputFile)
val words = input.flatMap(line => line.split(" "))
val word_counts = words.map(word => (word,
1)).reduceByKey{case (x, y) => x + y}
word_counts.saveAsTextFile(outputFile)
```

The Spark ecosystem



Spark Streaming

- Spark Streaming is built on top of Spark core engine and can be used to develop a fast, scalable, high throughput, and fault tolerant real-time system.
- Streaming data can come from any source, such as production logs, click-stream data, Kafka, Kinesis, Flume, and many other data serving systems.

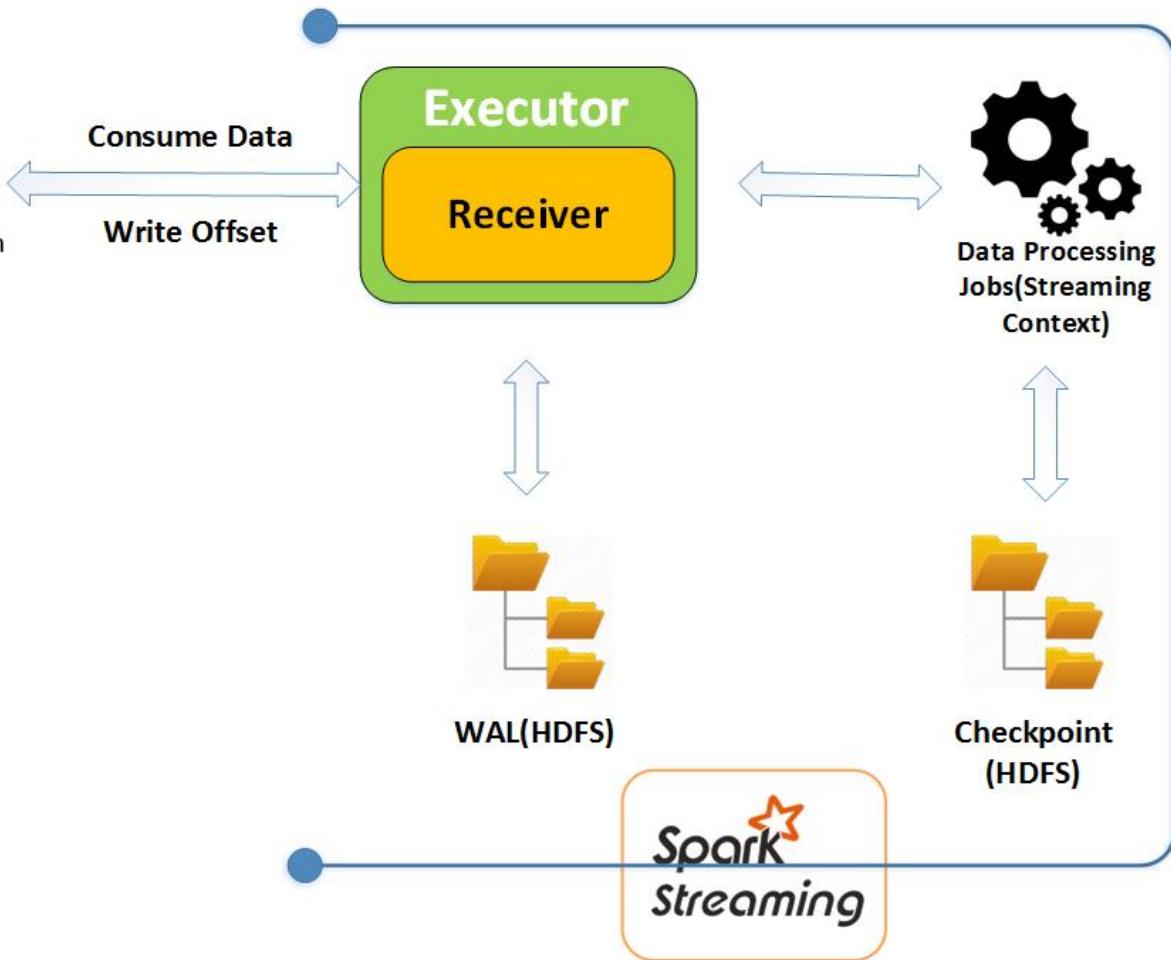
Spark Streaming

Basically, we have two approaches to integrate Kafka with Spark and we will go into detail on each:

- Receiver-based approach
- Direct approach

Receiver-based integration

- Spark uses Kafka high level consumer API to implement receiver.
- This is an old approach and data received from Kafka topic partitions are stored in Spark executors and processes by streaming jobs.
- However, Spark receiver replicates the message across all the executors, so that if one executor fails, another executor should be able to provide replicated data for processing.



Disadvantages of receiver-based approach

- **Throughput:** Enabling write-ahead log and checkpoint may cause you less throughput because time may be consumed in writing data to HDFS. It's obvious to have low throughput when there is lot of disk I/O involved.
- **Storage:** We store one set of data in Spark executor buffer and another set of the same data in write-ahead log HDFS. We are using two stores to store the same data and storage needs may vary, based on application requirements.
- **Data Loss:** If a write-ahead log is not enabled, there is a huge possibility of losing data and it may be very critical for some important applications.

Java example for receiver-based integration

- Let us take an example to be sure:

Refer to the file 5_1.txt

Scala example for receiver-based integration

- Here is an example on Scala:

Refer to the file 5_2.txt

Direct approach

- In receiver-based approach, we saw issues of data loss, costing less throughput using write-ahead logs and difficulty in achieving exactly one semantic of data processing.
- To overcome all these problems, Spark introduced the direct stream approach of integrating Spark with Kafka.



Fetch Latest Offset
(Topic – Partition)

Driver

Calculate Offset
Batch Range

Consume Data Based on
Calculated Offset Range

Executor

Spark
Streaming

- The following illustration gives a little detail about parallelism:

Java example for direct approach

- Again, let us take a Java example:

Refer to the file 5_3.txt

Scala example for direct approach

- Here is the Scala example for direct approach:

Refer to the file 5_4.txt

Use case log processing - fraud IP detection

- **Producer:** We will use Kafka Producer API, which will read a log file and publish records to Kafka topic. However, in a real case, we may use Flume or producer application, which directly takes a log record on a real-time basis and publish to Kafka topic.
- **Fraud IPs list:** We will maintain a list of predefined fraud IP range which can be used to identify fraud IPs. For this application we are using in memory IP list which can be replaced by fast key based lookup, such as HBase.
- **Spark Streaming:** Spark Streaming application will read records from Kafka topic and will detect IPs and domains which are suspicious.

Maven

- Maven is a build and project management tool and we will be building this project using Maven.
- I recommend using Eclipse or IntelliJ for creating projects.
- Add the following dependencies and plugins to your pom.xml:

Refer to the file 5_5.txt

Producer

- You can use IntelliJ or Eclipse to build a producer application.
- This producer reads a log file taken from an Apache project which contains detailed records like:

64.242.88.10 - - [08/Mar/2004:07:54:30 -0800] "GET
/twiki/bin/edit/Main/Unknown_local_recipient_reject_code
?topicparent>Main.ConfigurationVariables HTTP/1.1" 401
12846

Property reader

- We preferred to use a property file for some important values such as topic, Kafka broker URL, and so on.
- If you want to read more values from the property...

Summary

- In this lesson, we learned about Apache Spark, its architecture, and Spark ecosystem in brief.
- Our focus was on covering different ways we can integrate Kafka with Spark and their advantages and disadvantages.
- We also covered APIs for the receiver-based approach and direct approach.

Complete lab 5

6. Building Storm Applications with Kafka

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or development environment.

Building Storm Applications with Kafka

In this lesson, we will learn about:

- Introduction to Apache Storm
- Apache Storm architecture
- Brief overview of Apache Heron
- Integrating Apache Storm with Apache Kafka
(Java/Scala example)
- Use case (log processing)

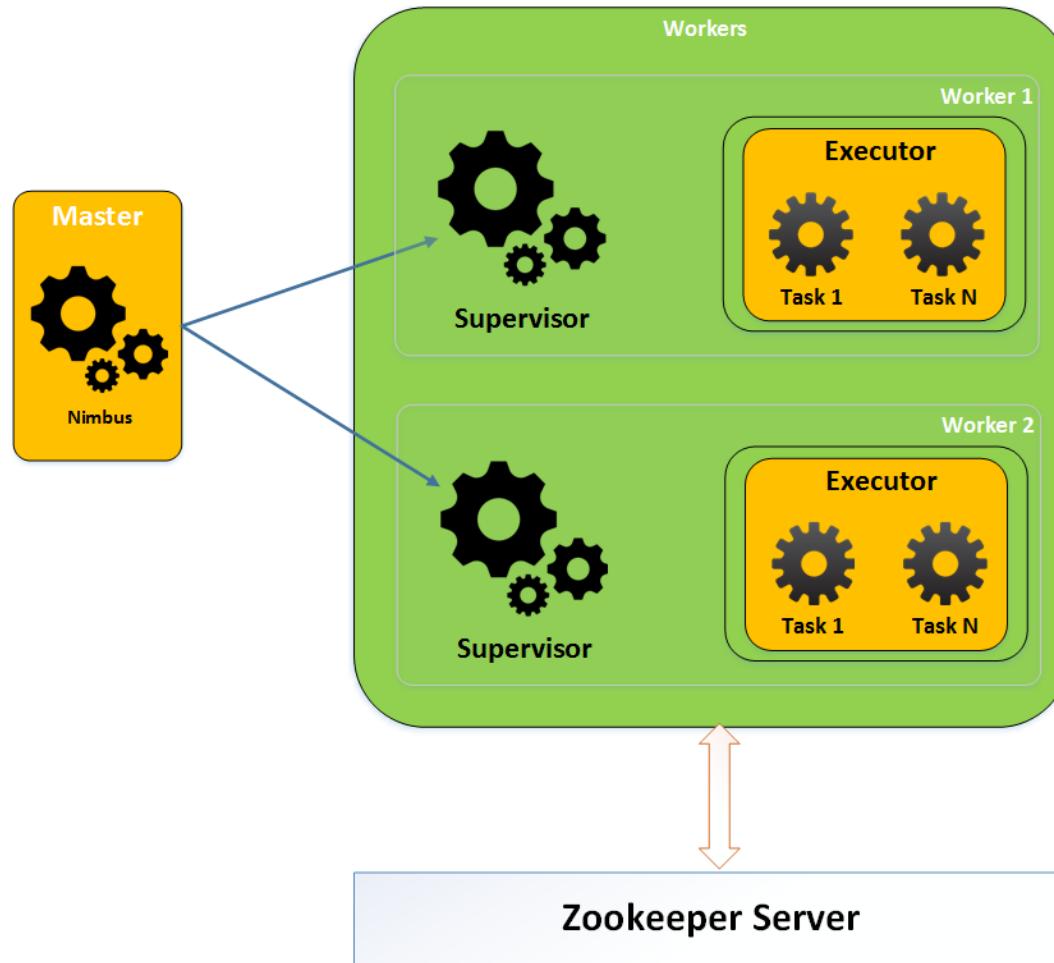
Introduction to Apache Storm

- Apache Storm is used to handle very sensitive applications where even a delay of 1 second can mean huge losses.
- There are many companies using Storm for fraud detection, building recommendation engines, triggering suspicious activity, and so on.
- Storm is stateless; it uses Zookeeper for coordinating purposes, where it maintains important metadata information.

Storm cluster architecture

Storm also follows the master-slave architecture pattern, where Nimbus is the master and Supervisors are the slaves:

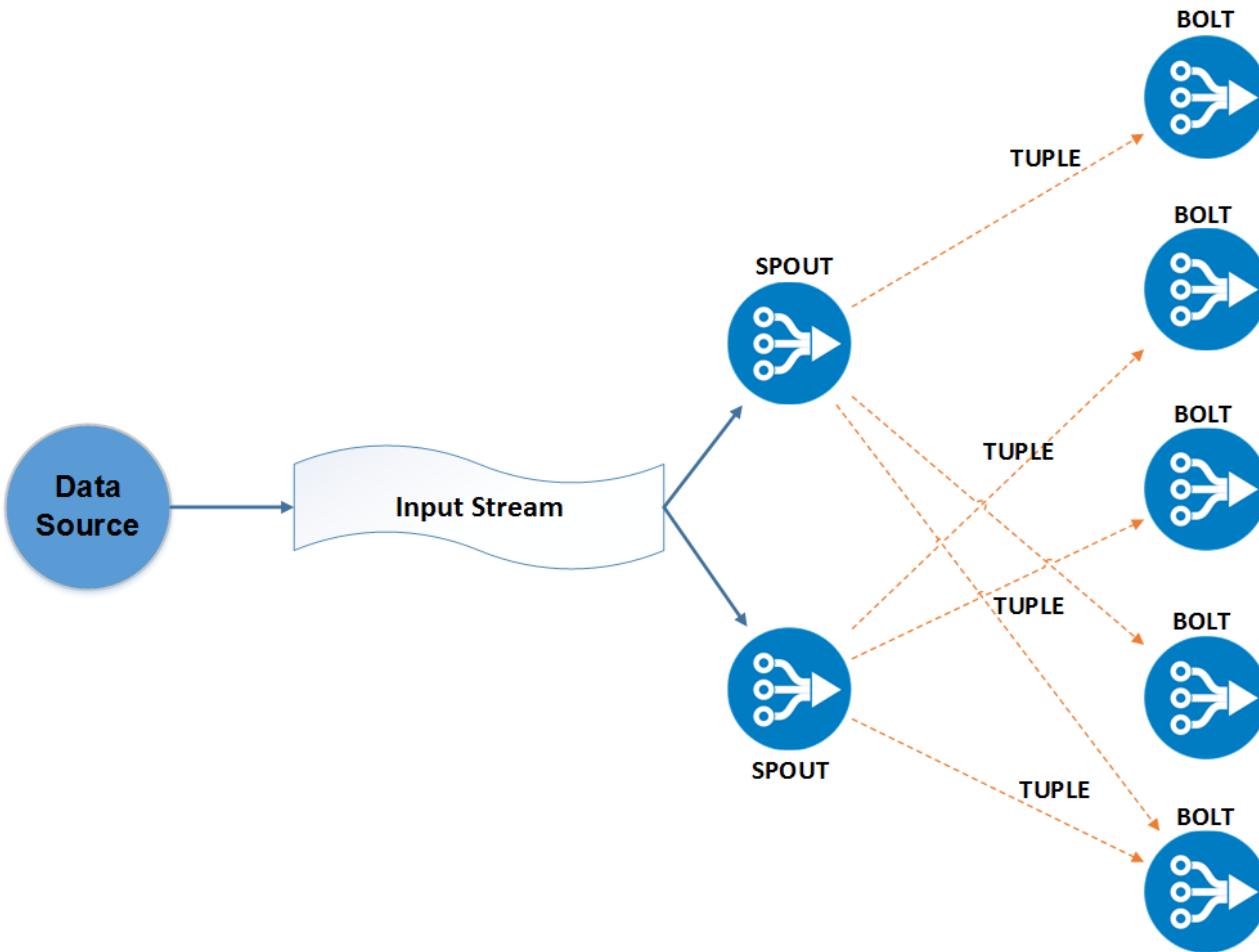
- Nimbus
- Supervisors



The concept of a Storm application

The Apache Storm application consists of two components:

- Spout
 - 1. Reliable spout
 - 2. Unreliable spout
- Bolt
- Topology

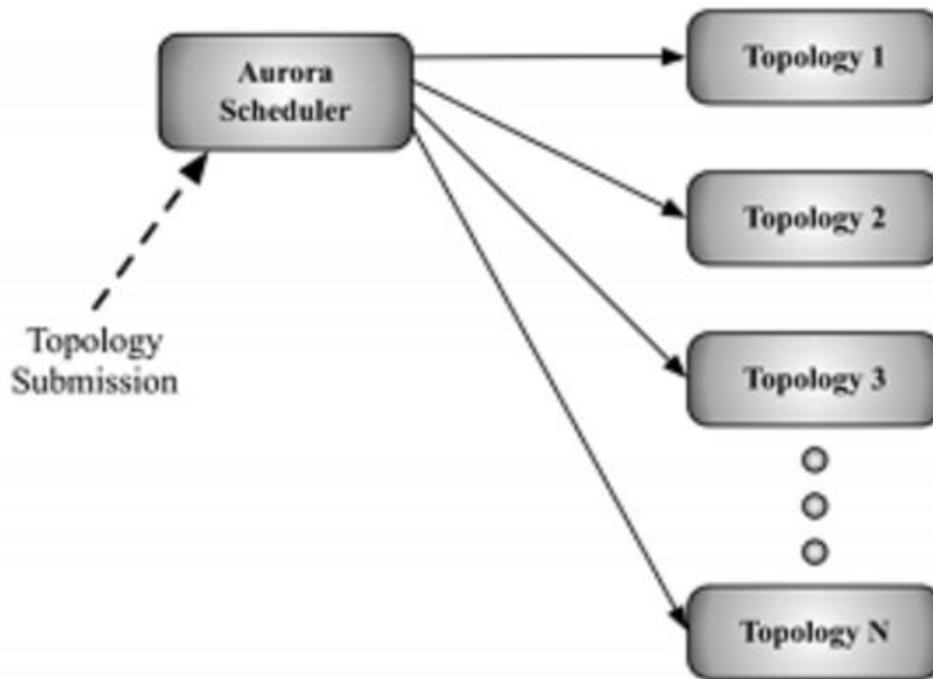


Introduction to Apache Heron

Apache Heron is the successor to Apache Storm with backward compatibility. Apache Heron provides more power in terms of throughput, latency, and processing capability over Apache Storm as use cases in Twitter started increasing, they felt of having new stream processing engine because of the following Storm bottleneck:

- Debugging
- Scale on Demand
- Cluster Manageability

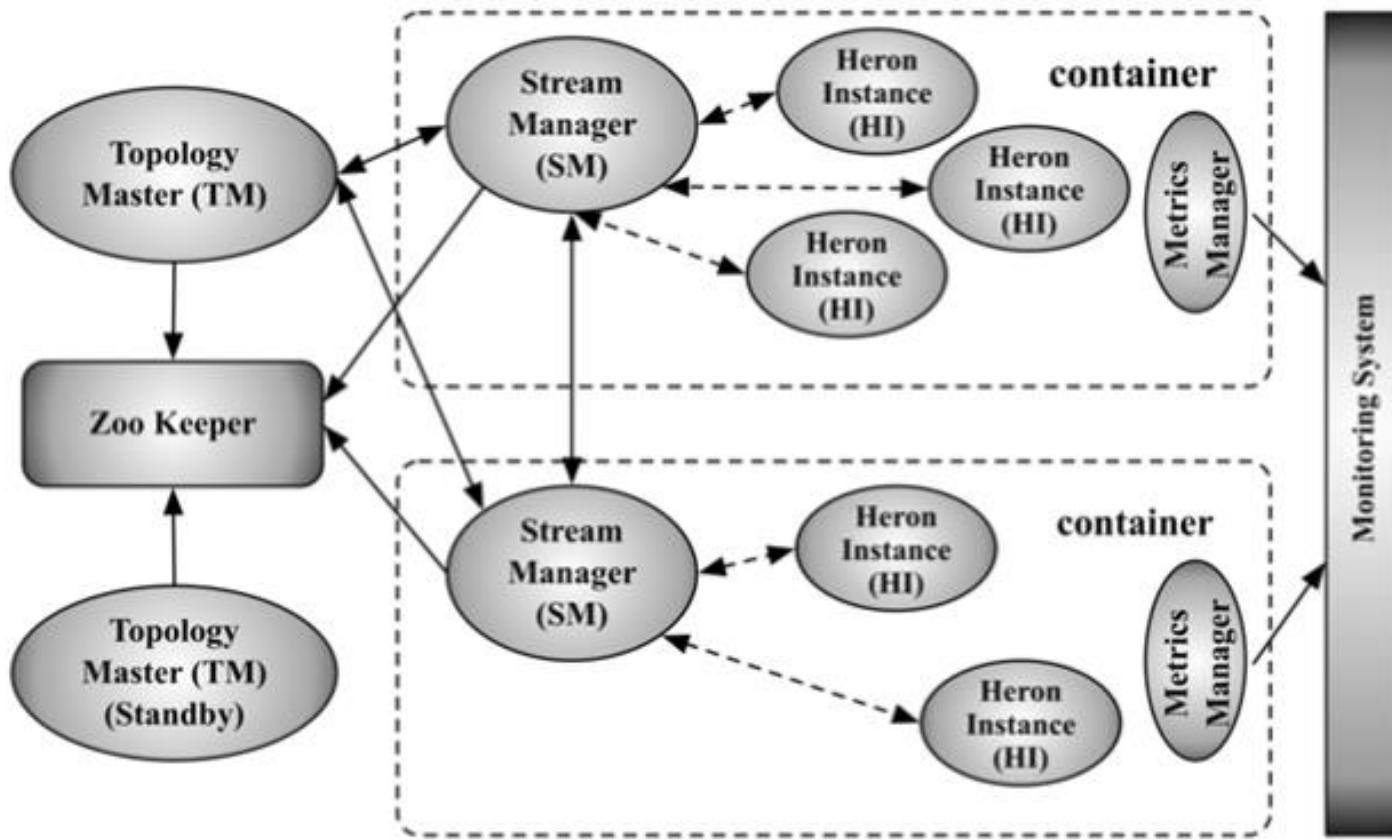
Heron architecture



Heron topology architecture

The following core components of Heron topology are discussed in depth in the sections as follows:

- Topology Master
- Container
- Stream Manager
- Heron Instance
- Metrics Manager
- Heron Tracker



Integrating Apache Kafka with Apache Storm - Java

- Look at the following code:

```
SpoutConfig spoutConfig = new SpoutConfig(hosts,  
inputTopic, "/" + zkRootDir, consumerGroup);  
spoutConfig.scheme = new  
SchemeAsMultiScheme(new StringScheme());  
spoutConfig.forceFromStart = false;  
spoutConfig.startOffsetTime =  
kafka.api.OffsetRequestLatestTime();
```

Integrating Apache Kafka with Apache Storm - Java

- Spout acts as a Kafka consumer and therefore it needs to manage the offset of records somewhere.
- Spout uses Zookeeper to store the offset, and the last two parameters in SpoutConfig denote the Zookeeper root directory path and ID for this particular spout.
- The offset will be stored as shown next, where 0, 1 are the partition numbers:

zkRootDir/consumerID/0

zkRootDir/consumerID/1

zkRootDir/consumerID/2

Integrating Apache Kafka with Apache Storm - Java

SchemeAsMultiScheme: It indicates how the ByteBuffer consumed from Kafka gets transformed into a Storm tuple.

- We have used the StringScheme implementation which will convert bytebuffer into string.
- Now the configuration is passed to KafkaSpout and the spout is set to topology:

```
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

Example

- Topology Class: The flow connection of spouts and bolts together forms a topology.
- In the following code, we have the TopologyBuilder class which allows us to set the connection flow:

```
TopologyBuilder topologyBuilder = new TopologyBuilder();
topologyBuilder.setSpout("kafkaspout", new
KafkaSpout(kafkaSpoutConfig));
topologyBuilder.setBolt("stringsplit", new
StringToWordsSpliterBolt()).shuffleGrouping("kafkaspout");
topologyBuilder.setBolt("counter", new
WordCountCalculatorBolt()).shuffleGrouping("stringsplit");
```

Example

- In the preceding code, we can see that spout is set to KafkaSpout and then kafkaspout is passed an input to string split bolt and splitbolt is passed to wordcount bolt.
- In this way, end to end topology pipeline gets created.

Refer to the file 6_1.txt

Example

- **String Split Bolt:** This is responsible for splitting lines into words and then transferring it to the next bolt in the topology pipeline:

Refer to the file 6_2.txt

Example

- Wordcount Calculator Bolt: It takes the input emitted by split bolt and then stores its count in Map, which finally gets dumped into console:

Refer to the file 6_3.txt

Integrating Apache Kafka with Apache Storm - Scala

- This section contains the Scala version of the wordcount program discussed previously.
- Topology Class: Let us try the topology class with Scala:

Refer to the file 6_4.txt

Integrating Apache Kafka with Apache Storm - Scala

- String Split Bolt: The same String Split Bolt on Scala:
Refer to the file 6_5.txt
- Wordcount Bolt: Example of Wordcount Bolt is given next:
Refer to the file 6_6.txt

Use case – log processing in Storm, Kafka, Hive

- Let us begin with the code and how it works. Copy the following classes from lesson 5, Building Spark Streaming Applications with Kafka, into your Storm Kafka use case:
- pom.xml:

Refer to the file 6_7.txt

Producer

- We will be reusing the producer code from the previous lesson.

streaming.properties file:

```
topic=iprecord
broker.list=10.200.99.197:6667
appname=IpFraud
group.id=Stream
```

Producer

- Property Reader:

Refer to the file 6_8.txt

Producer code

- We have also enabled auto creation of topics, so you need not create a topic before running your producer application.
- You can change the topic name in the streaming.properties file mentioned previously.

Refer to the file 6_9.txt

Fraud IP lookup

- We are using InMemoryLookup and have just added the fraud IP range in the cache.
- Add the following code to your project:

```
public interface IIPScanner {  
    boolean isFraudIP(String ipAddresses);  
}
```

Fraud IP lookup

- CacheIPLookup is one implementation for the IIPScanner interface which does in-memory lookup.

Refer to the file 6_10.txt

Storm application

- **Kafka Spout:** It will read a Stream of records from Kafka and will send it to two bolts
- **Fraud Detector Bolt:** This bolt will process the record emitted by Kafka spout and will emit fraud records to Hive and Kafka bolt
- **Hive Bolt:** This bolt will read the data emitted by fraud detector bolt and will process and push those records to hive table
- **Kafka Bolt:** This bolt will do the same processing as Hive bolt, but will push the resulting data to a different Kafka topic

Storm application

- `iptopology.properties:`

`zkhost = localhost:2181`

`inputTopic = iprecord`

`outputTopic=fraudip`

`KafkaBroker =localhost:6667`

`consumerGroup=id7`

`metaStoreURI = thrift://localhost:9083`

`dbName = default`

`tblName = fraud_ip`

Storm application

- Hive Table: Create the following table in hive; this table will store the records emitted by Hive bolt:

```
DROP TABLE IF EXISTS fraud_ip;
```

```
CREATE TABLE fraud_ip(
```

```
ip String,
```

```
date String,
```

```
request_url String,
```

```
protocol_type String,
```

```
status_code String
```

```
)
```

```
PARTITIONED BY (col1 STRING)
```

```
CLUSTERED BY (col3) into 5 buckets
```

```
STORED AS ORC;
```

Storm application

IPFraudDetectionTopology: This class will build the topology which indicates how spout and bolts are connected together to form Storm topology.

- This is the main class of our application and we will use it while submitting our topology to Storm cluster.

Refer to the file 6_11.txt

Storm application

Fraud Detector Bolt: This bolt will read the tuples emitted by Kafka spout and will detect which record is fraud by using an in memory IP lookup service.

- It will then emit the fraud records to hivebolt and kafkabolt simultaneously:

Refer to the file 6_12.txt

Storm application

- **IPFraudHiveBolt:** This call will process the records emitted by fraud detector bolt and will push the data to Hive using a thrift service:

Refer to the file 6_13.txt

Storm application

- **IPFraudKafkaBolt:** This uses the Kafka Producer API to push the processed fraud IP to another Kafka topic:

Refer to the file 6_14.txt

Running the project

- Execute the following permission before running the project:

```
sudo su - hdfs -c "hdfs dfs -chmod 777 /tmp/hive"  
sudo chmod 777 /tmp/hive
```

Running the project

- To run in cluster mode, we need to execute as:
Storm jar /home/ldap/chanchals/kafka-Storm-integration-0.0.1-SNAPSHOT.jar
com.fenago.Storm.ipfrauddetection.IPFraudDetectionTopology
iptopology.properties TopologyName
- To run in local mode, we need to execute as:
Storm jar kafka-Storm-integration-0.0.1-SNAPSHOT.jar
com.fenago.Storm.ipfrauddetection.IPFraudDetectionTopology
iptopology.properties

Summary

- In this lesson, we learned about Apache Storm architecture in brief and we also went through the limitations of Storm which motivated Twitter to develop Heron.
- We also discussed Heron architecture and its components.
- Later, we went through the API and an example of Storm Kafka integration.

Complete lab 6

7. Using Kafka with Confluent Platform

Using Kafka with Confluent Platform

We will walk through the following topics in this lesson:

- Introduction to Confluent Platform
- Confluent architecture
- Kafka Connectors and Kafka Streams
- Schema Registry and REST proxy
- Camus - moving Kafka data to HDFS

Introduction to Confluent Platform

The following are a few reasons to introduce Confluent Platform:

- Integration with Kafka
- In-built Connectors

Introduction to Confluent Platform

We provide another configuration, and the data is pushed to a destination such as Elasticsearch, HDFS and so on.

- Client
- Accessibility

Introduction to Confluent Platform

- **Storage format:** A common challenge that may occur where the producer and consumer of an application are loosely coupled with each other is data format.
- We may want to have a contract wherein we say any change in the data on the producer side should not affect all downstream consumer applications, or the producer should not accidentally produce data in a format that is not consumable by the consumer application.

Introduction to Confluent Platform

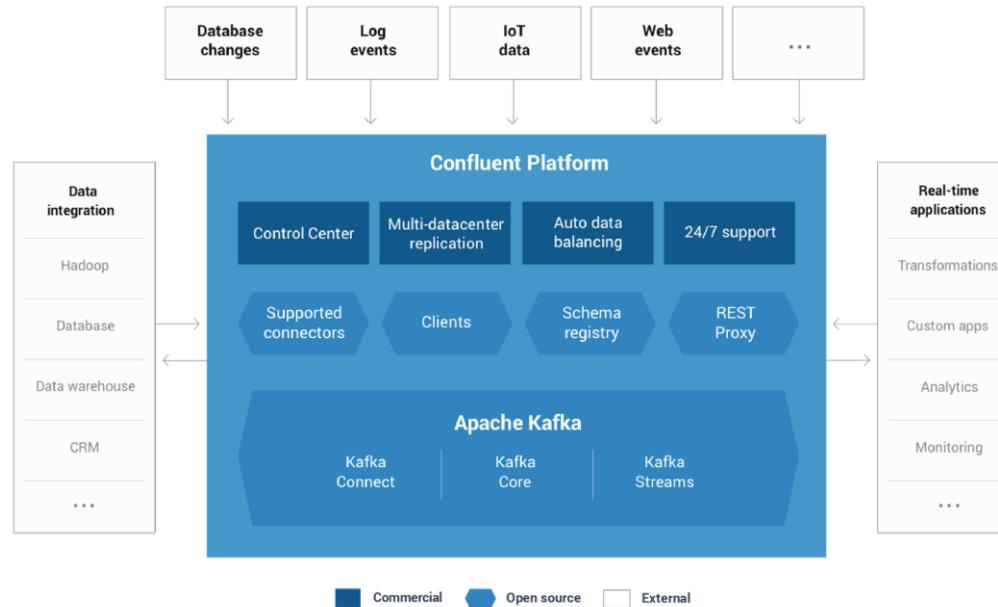
- Monitoring and controlling Kafka performance: We also want to have a mechanism where we can see the performance of the Kafka cluster, which should provide us with all the valuable metadata information with a good interface.
- we may want to see the performance of the topic, or we may want to see the CPU utilization of the Kafka cluster which may give us information about consumers at a deep level.

Deep driving into Confluent architecture

- The Confluent Platform provides you with underlying built-in Connectors and components that help you focus on the business use case and how to get value out of it.
- It takes care of the integration of data from multiple sources and its consumption by the target system.
- The Confluent Platform provides a trusted way of securing, monitoring, and managing the entire Kafka cluster infrastructure.

Deep driving into Confluent architecture

- The following image will give a concise idea of the Confluent architecture:



Understanding Kafka Connect and Kafka Stream

- Kafka Connect is a tool that provides the ability to move data into and out of the Kafka system.
- Thousands of use cases use the same kind of source and target system while using Kafka.
- Kafka Connect is comprised of Connectors for those common source or target systems.

Understanding Kafka Connect and Kafka Stream

Kafka Connect consists of a set of Connectors, and the Connectors are of two types:

- Import Connectors
- Export Connectors

Kafka Streams

- The processing engines require separate installation and maintenance efforts.
- Kafka Streams is a tool to process and analyze data stored in Kafka topics.
- The Kafka Stream library is built based on popular Stream processing concepts that allow you to run your streaming application on the Kafka cluster itself.

Playing with Avro using Schema Registry

- Schema Registry allows you to store Avro schemas for both producers and consumers.
- It also provides a RESTful interface for accessing this schema.
- It stores all the versions of Avro schema, and each schema version is assigned a schema ID.

Playing with Avro using Schema Registry

- Here is an example of Avro schema and producer:

```
kafka-avro-console-producer \
--broker-list localhost:9092 --topic test \
--property
value.schema='{"type":"record","name":"testrecord","fields":[{"name":"country","type":"string"}]}'
```

Playing with Avro using Schema Registry

- Similarly, an example of Avro schema on consumer:

```
kafka-avro-console-consumer --topic test\  
--Zookeeper localhost:2181 \  
--from-beginning
```

Playing with Avro using Schema Registry

- The schema can also be registered using a REST request as follows:

```
curl -X POST -H "Content-Type:  
application/vnd.schemaregistry.v1+json" \  
--data '{"schema": "{\"type\": \"string\"}"}' \  
http://localhost:8081/test/Kafka-test/versions  
{"id":1}
```

Moving Kafka data to HDFS

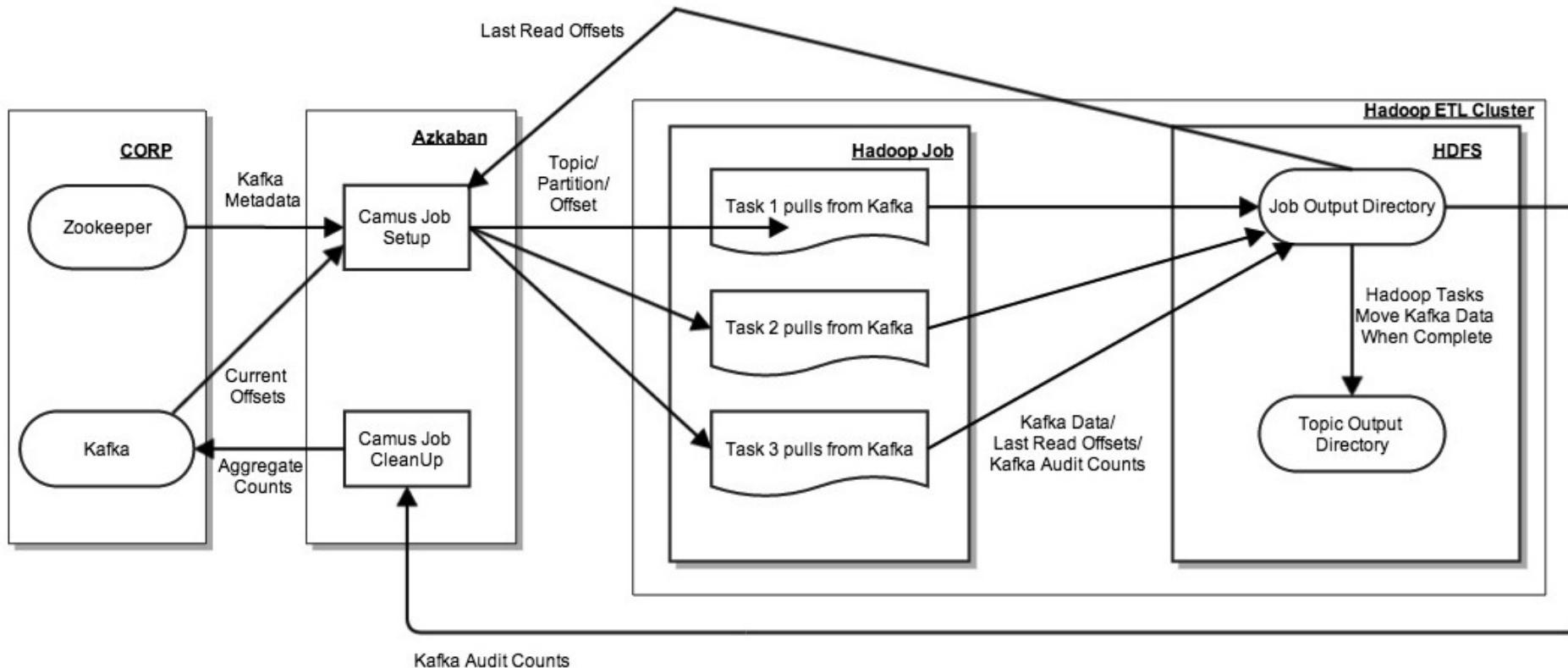
Kafka data can be moved to HDFS and can be used for different purposes. We will talk about the following four ways of moving data from Kafka to HDFS:

- Using Camus
- Using Gobblin
- Using Kafka Connect
- Using Flume

Camus

- LinkedIn first created Kafka for its own log processing use case.
- As discussed, Kafka stores data for a configured period of time, and the default is seven days.
- The LinkedIn team felt the need to store data for any batch-reporting purpose or to use it later.
- Now, to store data in HDFS, which is a distributed storage file system, they started developing a tool that can use a distributed system capability to fetch data from Kafka.

- The following image gives a good idea of the Camus architecture:



Running Camus

Camus mainly consists of two tasks:

- Reading data from Kafka
- Writing data to HDFS

Running Camus

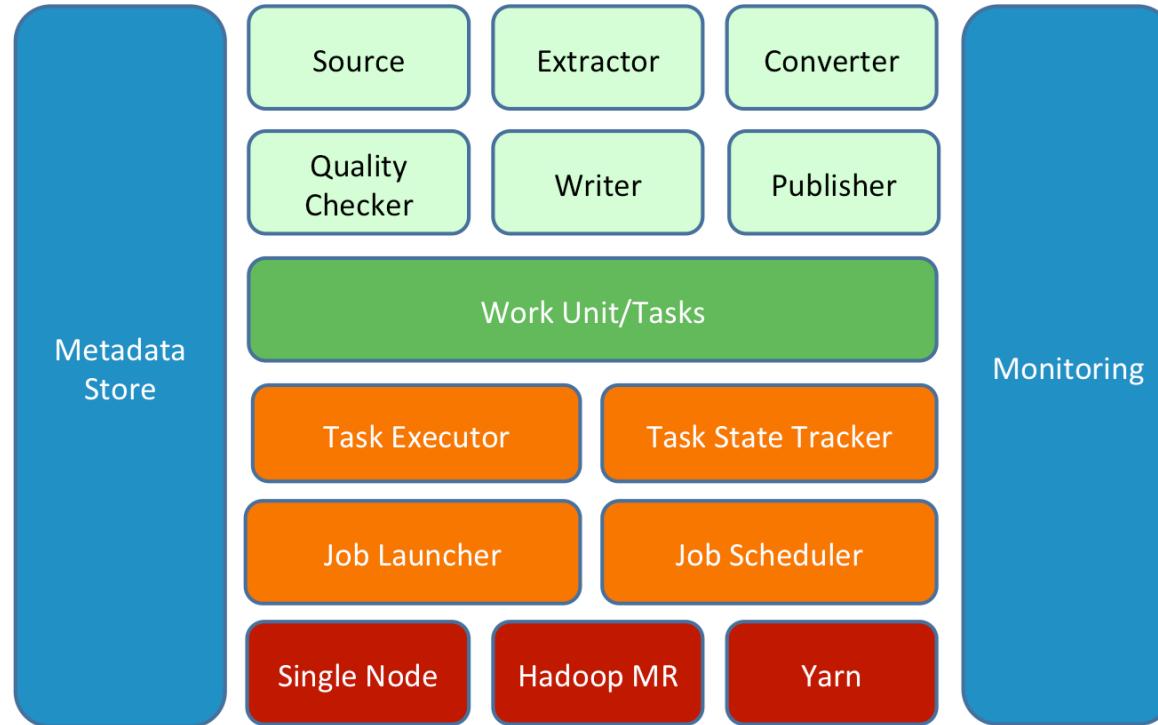
- Running Camus requires the Hadoop cluster. As discussed, Camus is nothing but a map-reduce job, and it can be run using normal Hadoop jobs as in the following case:

```
hadoop jar Camus.jar  
com.linkedin.Camus.etl.kafka.CamusJob -P  
Camus.properties
```

Gobblin

- Gobblin is an advanced version of Apache Camus. Apache Camus is only capable of copying data from Kafka to HDFS; however, Gobblin can connect to multiple sources and bring data to HDFS.
- LinkedIn had more than 10 data sources, and all were using different tools to ingest data for processing.

Gobblin architecture



Gobblin architecture

- The following configuration file (kafka_to_hdfs.conf) contains information about connection URLs, Sink type, output directory, and so on which will be read by Gobblin job to fetch data from Kafka to HDFS:

Refer to the file 7_1.txt

Gobblin architecture

- Run Gobblin-MapReduce.sh:

```
Gobblin-MapReduce.sh --conf kafka_to_hdfs.conf
```

Kafka Connect

- We already discussed Kafka Connect in the preceding sections.
- Kafka Connect refers to the Connectors that can be used to import or export data from Kafka.
- Kafka HDFS export Connector can be used to copy data from Kafka topic to HDFS.
- HDFS Connector polls data from Kafka and writes them to HDFS.

Kafka Connect

- Here is an example of Kafka Connect with producer:
`kafka-avro-console-producer --broker-list localhost:9092 --topic test \
--property
value.schema='{"type":"record","name":"peoplerecord","fields": [{"name":
:"f1","type":"string"}]}'`
- Run `kafka_to_hdfs.properties`:
`name=hdfs-
sinkConnector.class=io.confluent.connect.hdfs.HdfsSinkConnectortask
s.max=1topics=testhdfs.url=hdfs://localhost:8020flush.size=3`

Kafka Connect

- Run the following:

```
connect-standalone etc/schema-registry/connect-avro-  
standalone.properties \  
kafka_to_hdfs.properties
```

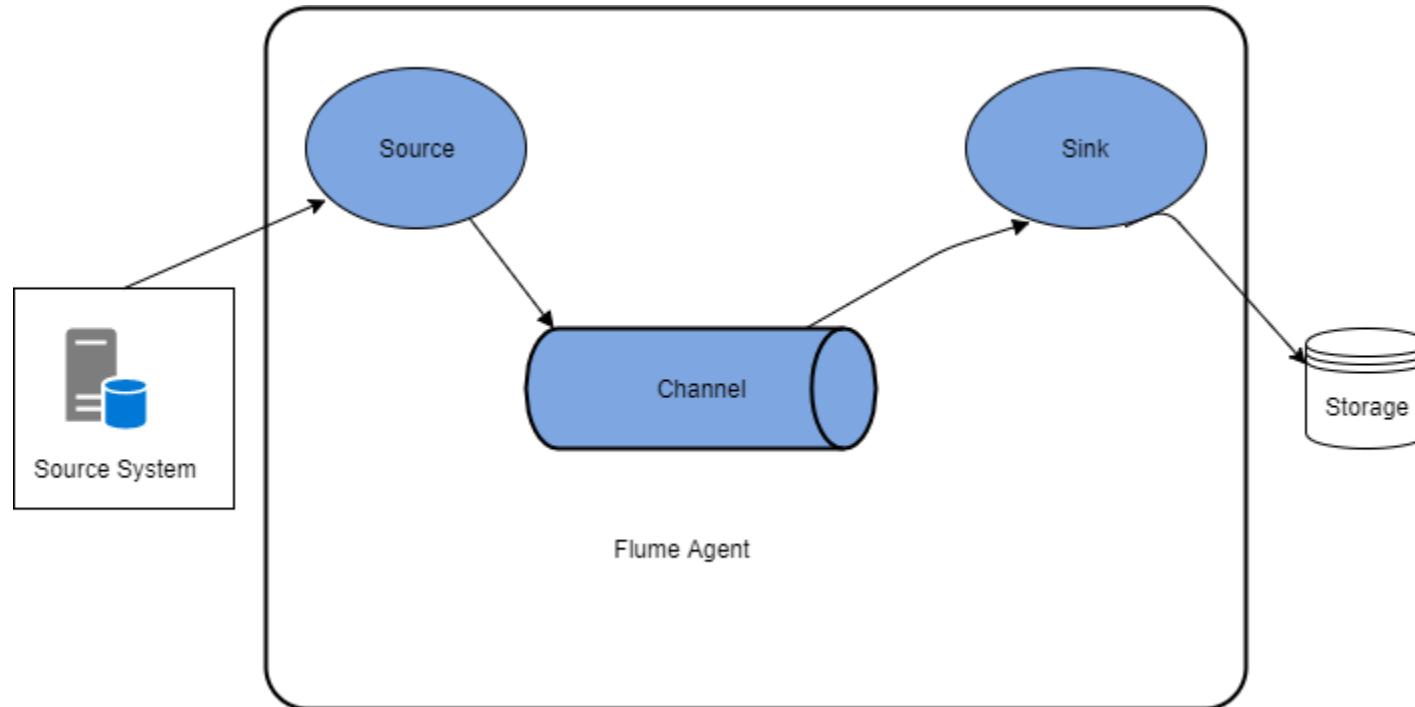
Flume

Apache Flume is a distributed, reliable, and fault-tolerant system for collecting large volumes of data from different sources to one or more target systems.

It mainly consists of three components:

- Source
- Channel
- Sink

Flume



Flume

- Let us first look into flumekafka.conf:

Refer to the file 7_2.txt

Flume

- The following configuration connects the source with the channel:

```
pipeline.sources.kafka1.channels = channel1
```

- The following configuration connects the sink with the channel:

```
pipeline.sinks.hdfssink.channel = channel1
```

Flume

- pipeline is an agent name, which you can change according to your wish.
- Once the agent configuration is ready, we can run Flume using the following command:

```
flume-ng agent -c pathoflume/etc/flume-ng/conf -f  
flumekafka.conf -n pipeline
```

Summary

- This lesson has given us a brief understanding of Confluent Platform and it uses.
- You learned about the architecture of Confluent Platform and how Connectors can make our job of transporting data in and out of Kafka simpler.
- We also learned about how the Schema Registry solves data format issues and supports schema resolution.

Complete lab 7

8. Building ETL Pipelines Using Kafka

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or data processing environment.

Building ETL Pipelines Using Kafka

In this lesson, we will cover Kafka Connect in detail. The following are the topics we will cover:

- Use of Kafka in the ETL pipeline
- Introduction to Kafka Connect
- Kafka Connect architecture
- Deep dive into Kafka Connect
- Introductory example of Kafka Connect
- Common use cases

Considerations for using Kafka in ETL pipelines

ETL is a process of Extracting, Transforming, and Loading data into the target system, which is explained next. It is followed by a large number of organizations to build their data pipelines.

- Extraction
- Transformation
- Loading

Considerations for using Kafka in ETL pipelines

Let's look into how we can use Kafka in an ETL operation:

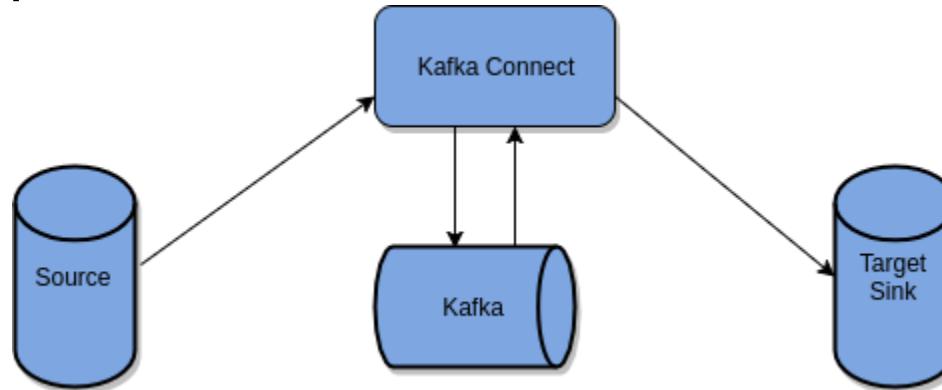
- Working of extracting operation of ETL
- Working of transforming operation of ETL

Introducing Kafka Connect

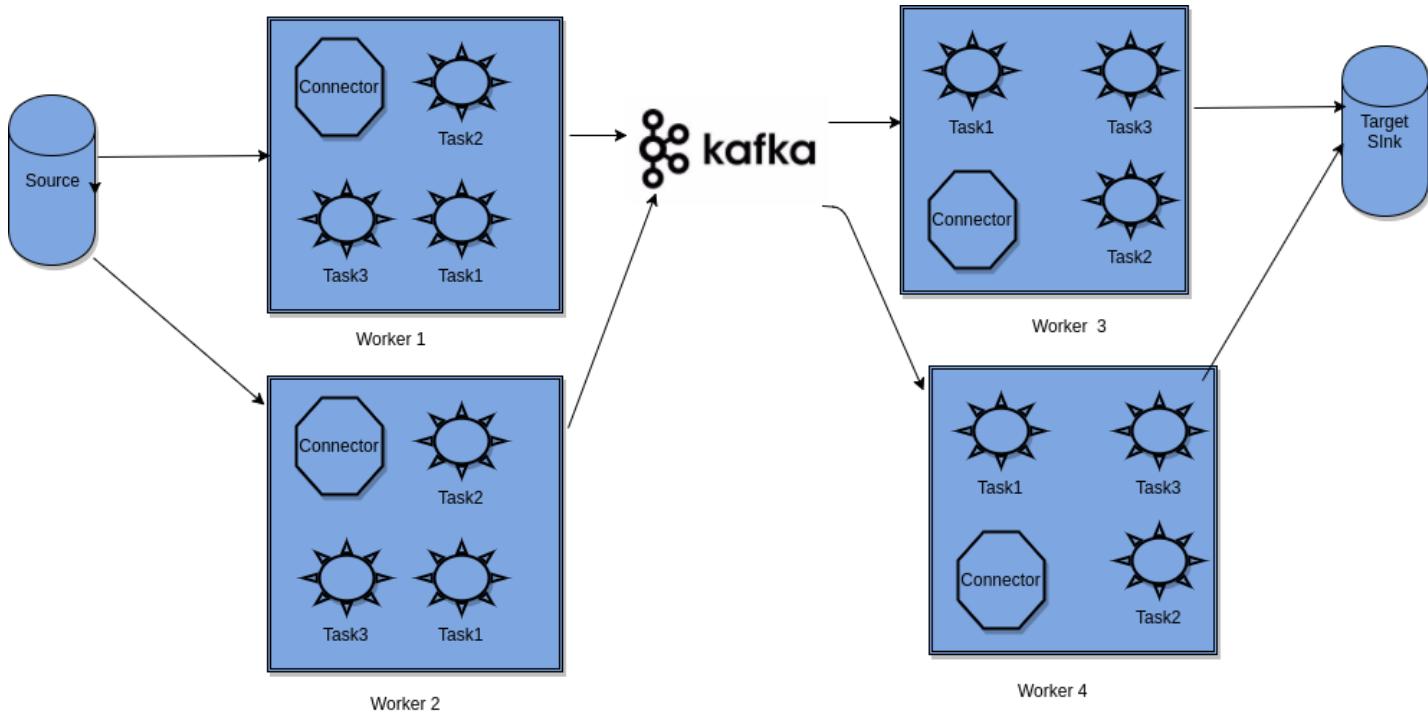
- Kafka Connect is used to copy data into and out of Kafka.
- There are already a lot of tools available to move data from one system to another system.
- You would find many use cases where you want to do real-time analytics and batch analytics on the same data.
- Data can come from different sources but finally may land into the same category or type.

Introducing Kafka Connect

- We may want to bring this data to Kafka topics and then pass it to a real-time processing engine or store it for batch processing.
- If you closely look at the following figure, there are different processes involved:



Deep dive into Kafka Connect



Introductory examples of using Kafka Connect

We have tested the code on the Ubuntu machine. Download the Confluent Platform tar file from the Confluent website:

- **Import or Source Connector:** This is used to ingest data from the source system into Kafka. There are already a few inbuilt Connectors available in the Confluent Platform.
- **Export or Sink Connector:** This is used to export data from Kafka topic to external sources. Let's look at a few Connectors available for real-use cases.
- **JDBC Source Connector:** The JDBC Connector can be used to pull data from any JDBC-supported system to Kafka.

Introductory examples of using Kafka Connect

- Install sqllite:

```
sudo apt-get install sqlite3
```

- Start console:

```
sqlite3 fenago.db
```

Introductory examples of using Kafka Connect

- Create a database table, and insert records:

```
sqlite> CREATE TABLE authors(id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name VARCHAR(255));
```

```
sqlite> INSERT INTO authors(name) VALUES('Manish');
```

```
sqlite> INSERT INTO authors(name) VALUES('Chanchal');
```

Introductory examples of using Kafka Connect

- Make the following changes in the source-quickstart-sqlite.properties file:

```
name=jdbc-
testConnector.class=io.confluent.connect.jdbc.JdbcSource
Connector.tasks.max=1
connection.url=jdbc:sqlite:fenga
dbmode=incrementing
incrementing.column.name=id
topic.prefix=test-
```

Introductory examples of using Kafka Connect

- In connection.url, the fenago.db value is the path to your fenago.db file. Provide the full path to the .db file.
- Once everything is ready, run the following command to execute the Connector script:

```
./bin/connect-standalone etc/schema-registry/connect-avro-standalone.properties etc/kafka-connect-jdbc/source-quickstart-sqlite.properties
```

Introductory examples of using Kafka Connect

- Once the script is successfully executed, you can check the output using the following command:

```
bin/kafka-console-consumer --new-consumer --  
bootstrap-server localhost:9092 --topic test-authors --  
from-beginning
```

Introductory examples of using Kafka Connect

- You will see the following output:

```
SLF4J: Class path contains multiple SLF4J bindings.  
SLF4J: Found binding in [jar:file:/home/chanchal/projects/confluent-3.2.2/share/ja  
StaticLoggerBinder.class]  
SLF4J: Found binding in [jar:file:/home/chanchal/projects/confluent-3.2.2/share/ja  
aticLoggerBinder.class]  
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.  
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]  
{"id":1,"name": {"string": "Manish"} }  
{"id":2,"name": {"string": "Chanchal"} }  
[
```

Introductory examples of using Kafka Connect

- Configure sink-quickstart-sqlite.properties:

```
name=test-jdbc-
sinkConnector.class=io.confluent.connect.jdbc.JdbcSink
Connectortasks.max=1topics=authors_sinkconnection.u
rl=jdbc:sqlite:fenago_authors.dbauto.create=true
```

Introductory examples of using Kafka Connect

- Run the producer:

```
bin/kafka-console-producer \
--broker-list localhost:9092 --topic authors_sink \
--property
value.schema='{"type":"record","name":"authors","fields":[{"name": "id", "type": "int"}, {"name": "author_name", "type": "string"}, {"name": "age", "type": "int"}, {"name": "popularity_percentage", "type": "float"}]}'
```

Introductory examples of using Kafka Connect

- Run the Kafka Connect Sink:

```
./bin/connect-standalone etc/schema-registry/connect-avro-  
standalone.properties etc/kafka-connect-jdbc/sink-quickstart-  
sqlite.properties
```

- Insert the record into the producer:

```
{"id": 1, "author_name": "Chanchal", "age": 26,  
"popularity_percentage": 60}  
{"id": 2, "author_name": "Manish", "age": 32,  
"popularity_percentage": 80}
```

Introductory examples of using Kafka Connect

- Run sqlite:

```
sqlite3 fenago_authors.db  
select * from authors_sink;
```

- You will see following output in the table:

```
sqlite> select * from authors_sink;  
Chanchal|60.0|1|26  
Manish|80.0|2|32  
sqlite> █
```

Kafka Connect common use cases

Let's understand a few common use cases of Kafka Connect:

- Copying data to HDFS
- Replication
- Importing database records
- Exporting Kafka records

Summary

- In this lesson, we learned about Kafka Connect in detail. We also learned about how we can explore Kafka for an ETL pipeline.
- We covered examples of JDBC import and export Connector to give you a brief idea of how it works.
- We expect you to run this program practically to get more insight into what happens when you run Connectors.

Complete lab 8

9. Building Streaming Applications Using Kafka Streams

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or programming environment.

Building Streaming Applications Using Kafka Streams

We will cover the following topics in this lesson:

- Introduction to Kafka Stream
- Kafka Stream architecture
- Advantages of using Kafka Stream
- Introduction to KStream and KTable
- Use case example

Introduction to Kafka Streams

The data processing strategy has evolved over time, and it's still being used in different ways. The following are the important terms related to Kafka Streams:

- Request/response
- Batch processing
- Stream processing

Using Kafka in Stream processing

Kafka is the persistence queue for data where data is stored in order of time stamp. The following properties of Kafka allow it to occupy its place in most Streaming architecture:

- Persistence queue and loose coupling
- Fault tolerance
- Logical ordering
- Scalability

Kafka Stream - lightweight Stream processing library

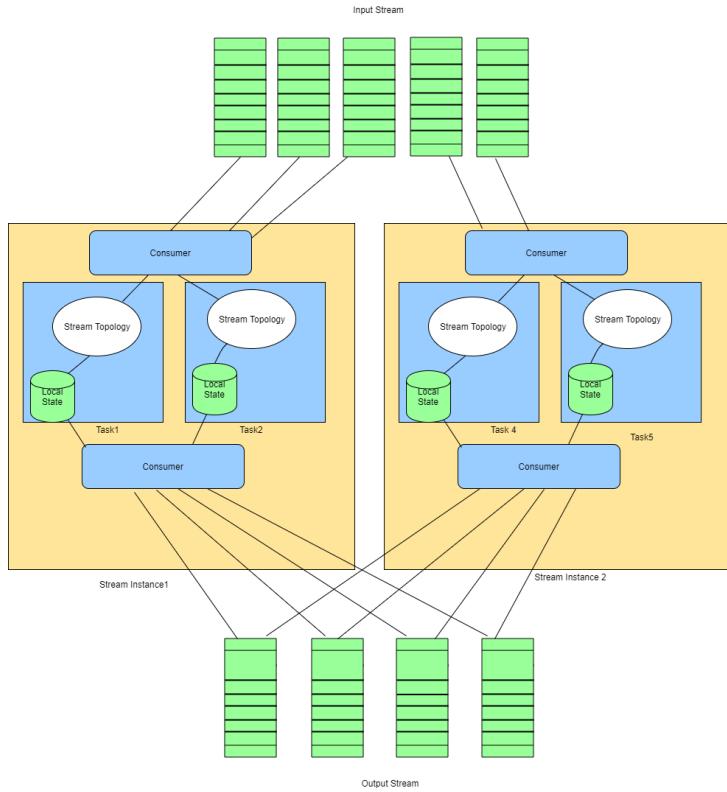
- Kafka Stream is a lightweight Stream processing library that is tightly coupled with Kafka. It does not require any cluster setup or any other operational cost.
- We will discuss the features that any Stream processing application should persist and how Kafka Stream provides those features.

Kafka Stream architecture

- Kafka Streams internally uses the Kafka producer and consumer libraries.
- It is tightly coupled with Apache Kafka and allows you to leverage the capabilities of Kafka to achieve data parallelism, fault tolerance, and many other powerful features.

Kafka Stream architecture

- The following figure is an internal representation of the working of Kafka Stream:



Integrated framework advantages

Deployment: An application built using Kafka Stream does not require any extra setup of the clusters to run. It can be run from a single-node machine or from your laptop.

- This is a huge advantage over other processing tools, such as Spark, Storm, and so on, which require clusters to be ready before you can run the application.
- Kafka Stream uses Kafka's producer and consumer library.

Integrated framework advantages

You need to specify the details of the Kafka cluster when you write your Stream application.

- Simple and easy features
- Coordination and fault tolerance

Understanding tables and Streams together

Maven dependency

- The Kafka Stream application can be run from anywhere, You just need to add library dependency and start developing your program.
- We are using Maven to build our application, Add the following dependency into your project:

```
<dependency>
    <groupId>org.apache.Kafka</groupId>
    <artifactId>Kafka-Streams</artifactId>
    <version>0.10.0.0</version>
</dependency>
```

Kafka Stream word count

- The following code is a simple word count program built using a Stream API.
- We will go through the important APIs used in this program, and will talk about their uses:

Refer to the file 9_1.txt

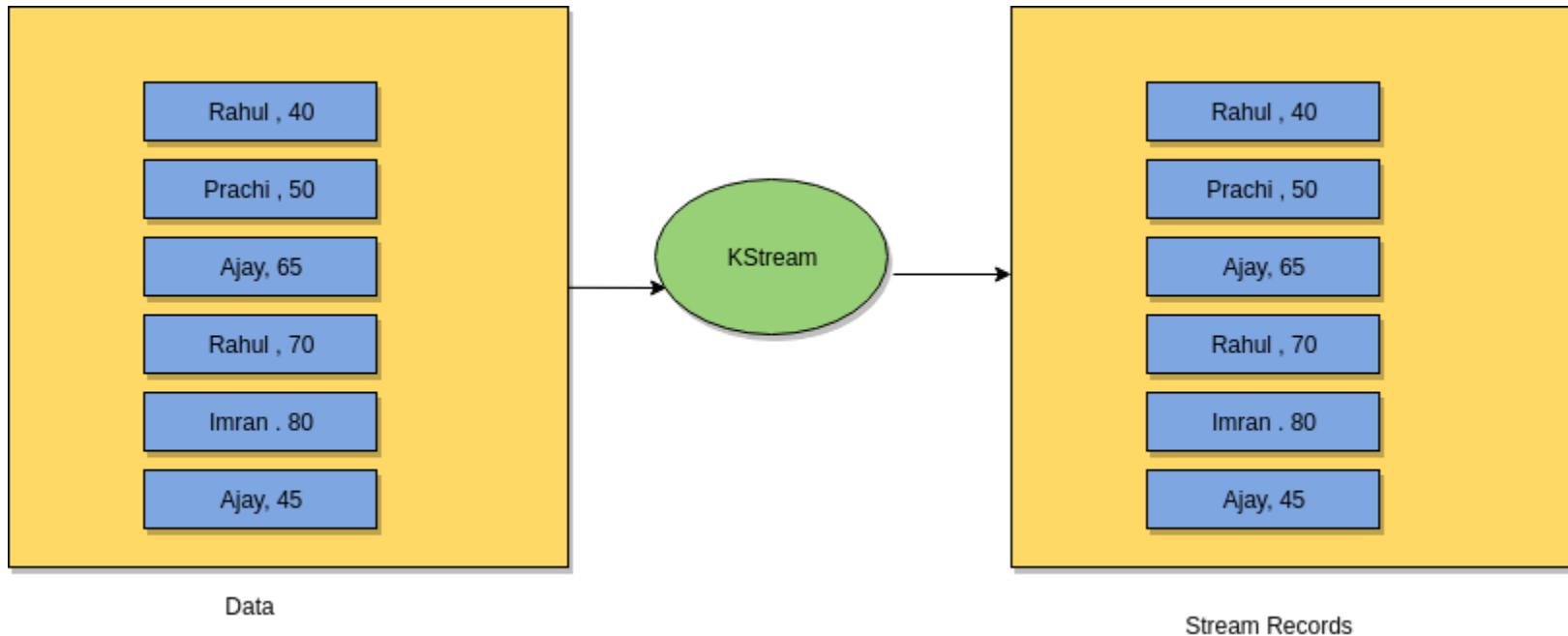
Kafka Stream word count

- Using the Kafka topic: Any Kafka Stream application starts with KStream, which consumes data from the Kafka topic.
- If you look into the earlier program, the following lines create KStream topicRecords, which will consume data from the topic input:

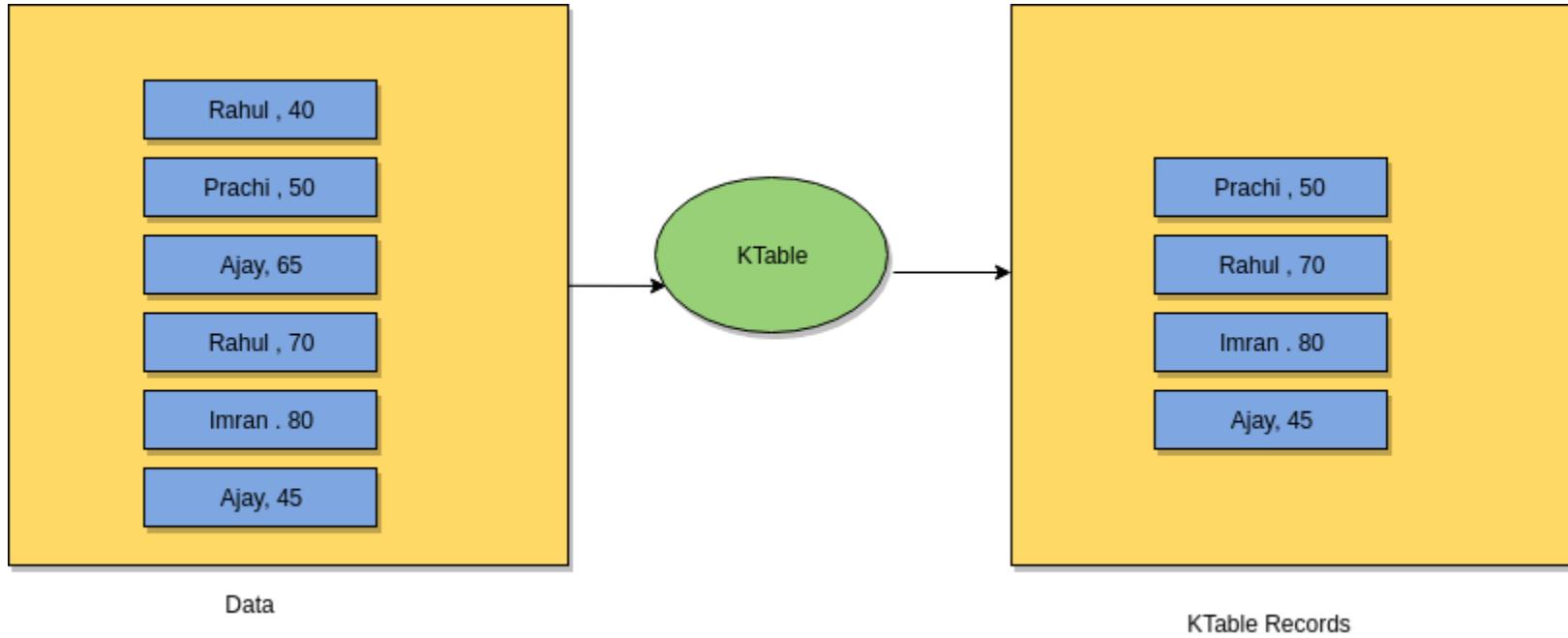
```
KStream<String, String> topicRecords =  
    StreamTopology.Stream(stringSerde, stringSerde, "input");
```

Kafka Stream word count

- Using transformation



KTable



Use case example of Kafka Streams

- Let's start with how we can build the same application using Kafka Stream.
- We will start with the code, take the producer, and look up the code from lesson 6, Building Storm Application with Kafka, which can be utilized here as well.

Maven dependency of Kafka Streams

- The best part of Kafka Stream is that it does not require any extra dependency apart from Stream libraries.
- Add the dependency to your pom.xml:

Refer to the file 9_2.txt

Property reader

- We are going to use the same property file and property reader that we used in lesson 6, Building Storm Application with Kafka, with a few changes.
- Kafka Stream will read the record from the iprecord topic and will produce the output to the fraudIp topic:

topic=iprecord
broker.list=localhost:9092
output_topic=fraudIp

Property reader

- Here is the property reader class:

Refer to the file 9_3.txt

IP record producer

- The producer will auto-create the topic if it does not exist.
- Here is how the code goes:

Refer to the file 9_4.txt

IP record producer

- Verify the producer record using the console producer.
- Run the following command on the Kafka cluster:

Kafka-console-consumer --zookeeper localhost:2181 --topic iprecord --from-beginning

IP record producer

```
Using the ConsoleConsumer with old consumer is deprecated and will be removed in a future major release. Consider using the new consumer
sing [bootstrap-server] instead of [zookeeper].
49.10.237.128, -, -, 07/Mar/2004:16:05:49, -0800, "GET, /twiki/bin/edit/Main/Double_bounce_sender?topicparent>Main.ConfigurationVariables
/1.1", 401, 12846
169.100.71.241, -, -, 07/Mar/2004:16:06:51, -0800, "GET, /twiki/bin/rdiff/TWiki/NewUserTemplate?rev1=1.3&rev2=1.2, HTTP/1.1", 200, 4523
90.131.75.45, -, -, 07/Mar/2004:16:10:02, -0800, "GET, /mailman/listinfo/hsdivision, HTTP/1.1", 200, 6291
58.202.218.174, -, -, 07/Mar/2004:16:11:58, -0800, "GET, /twiki/bin/view/TWiki/WikiSyntax, HTTP/1.1", 200, 7352
202.63.36.26, -, -, 07/Mar/2004:16:20:55, -0800, "GET, /twiki/bin/view/Main/DCCAndPostFix, HTTP/1.1", 200, 5253
18.250.52.72, -, -, 07/Mar/2004:16:23:12, -0800, "GET, /twiki/bin/oops/TWiki/AppendixFileSystem?template=oopsmore&param1=1.12&param2=1.12
/1.1", 200, 11382
59.90.177.88, -, -, 07/Mar/2004:16:24:16, -0800, "GET, /twiki/bin/view/Main/PeterThoeny, HTTP/1.1", 200, 4924
0.64.150.25, -, -, 07/Mar/2004:16:29:16, -0800, "GET, /twiki/bin/edit/Main/Header_checks?topicparent>Main.ConfigurationVariables, HTTP/1.
1, 12851
76.148.132.191, -, -, 07/Mar/2004:16:30:29, -0800, "GET, /twiki/bin/attach/Main/OfficeLocations, HTTP/1.1", 401, 12851
139.133.221.180, -, -, 07/Mar/2004:16:31:48, -0800, "GET, /twiki/bin/view/TWiki/WebTopicEditTemplate, HTTP/1.1", 200, 3732
67.44.110.109, -, -, 07/Mar/2004:16:32:50, -0800, "GET, /twiki/bin/view/Main/WebChanges, HTTP/1.1", 200, 40520
55.24.117.232, -, -, 07/Mar/2004:16:33:53, -0800, "GET, /twiki/bin/edit/Main/Smtpd_etrn_restrictions?topicparent>Main.ConfigurationVariab
LTP/1.1", 401, 12851
122.232.93.84, -, -, 07/Mar/2004:16:35:19, -0800, "GET, /mailman/listinfo/business, HTTP/1.1", 200, 6379
139.232.241.115, -, -, 07/Mar/2004:16:36:22, -0800, "GET, /twiki/bin/rdiff/Main/WebIndex?rev1=1.2&rev2=1.1, HTTP/1.1", 200, 46373
98.74.200.147, -, -, 07/Mar/2004:16:37:27, -0800, "GET, /twiki/bin/view/TWiki/DontNotify, HTTP/1.1", 200, 4140
48.220.6.25, -, -, 07/Mar/2004:16:39:24, -0800, "GET, /twiki/bin/view/Main/TokyoOffice, HTTP/1.1", 200, 3853
220.105.174.45, -, -, 07/Mar/2004:16:43:54, -0800, "GET, /twiki/bin/view/Main/MikeMannix, HTTP/1.1", 200, 3686
```

IP lookup service

- The IP scanner interface looks like this:

```
package com.fenago.Kafka.lookup;  
  
public interface IIPScanner {  
  
    boolean isFraudIP(String ipAddresses);  
  
}
```

IP lookup service

- We have kept the in-memory IP lookup very simple for an interactive execution of the application.
- The lookup service will scan the IP address and detect whether the record is a fraud or not by comparing the first 8 bits of the IP address:

Refer to the file 9_5.txt

Fraud detection application

- The fraud detection application will be running continuously, and you can run as many instances as you want; Kafka will do the load balancing for you.
- Let's look at the following code that reads the input from the iprecord topic and then filters out records that are fraud using the lookup service:

Refer to the file 9_6.txt

Summary

- In this lesson, you learned about Kafka Stream and how it makes sense to use Kafka Stream to do transformation when we have Kafka in our pipeline.
- We also went through the architecture, internal working, and integrated framework advantages of Kafka Streams.

Complete lab 9

10. Kafka Cluster Deployment



Kafka Cluster Deployment

In this lesson, we will cover the following topics:

- Kafka cluster internals
- Capacity planning
- Single-cluster deployment
- Multi-cluster deployment
- Decommissioning brokers
- Data migration

Kafka cluster internals

- Well, this topic has been covered in bits and pieces in the introductory lessons of this course.
- However, in this section, we are covering this topic with respect to components or processes that play an important role in Kafka cluster.

Role of Zookeeper

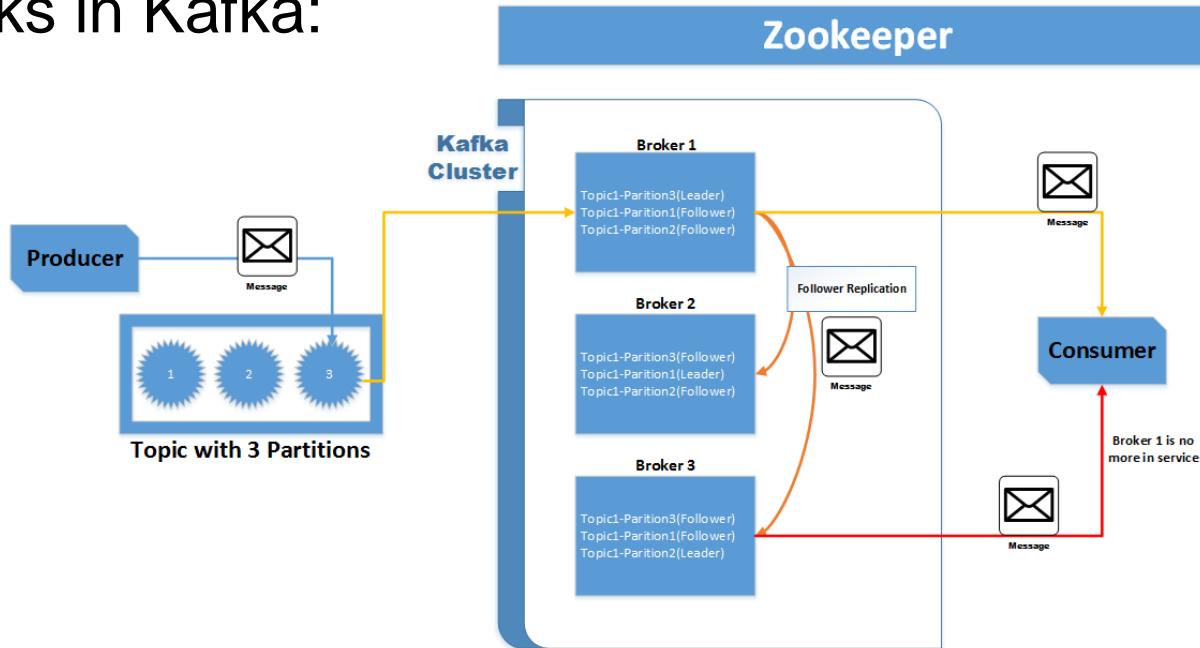
- Kafka clusters cannot run without Zookeeper servers, which are tightly coupled with Kafka cluster installations.
- Therefore, you should first start this section by understanding the role of Zookeeper in Kafka clusters.
- If we must define the role of Zookeeper in a few words, we can say that Zookeeper acts a Kafka cluster coordinator that manages cluster membership of brokers, producers, and consumers participating in message transfers via Kafka.

Replication

- One of the important aspects of Kafka is that it is highly available, and this is guaranteed through data replication.
- Replication is the core principle of Kafka design.
- Any type of client (producer or consumer) interacting with Kafka cluster is aware of the replication mechanism implemented by Kafka.

Replication

- The diagram following represents how replication works in Kafka:



Metadata request processing

- Before we jump into producer or consumer request processing, we should understand some of the common activities that any Kafka client or broker would perform irrespective of whether it is a write request or fetch request.
- One such request is to understand how metadata is requested or fetched by Kafka clients.

Producer request processing

All the write requests contain a parameter called ack, which determines when brokers should respond with a successful write to the client. Following are the possible values of the ack configuration:

- 1: This means the message is accepted only by the leader
- all: This means all in-sync replicas along with the leader have accepted the message
- 0: This means do not wait for acceptance from any of the brokers

Consumer request processing

- Same as the producer requests, consumer fetch requests start with metadata requests.
- Once the consumer is aware of the leader information, it forms a fetch request containing an offset from which it wants to read the data.
- It also provides the minimum and maximum number of messages it wants to read from the leader broker.

Capacity planning

- Capacity planning is mostly required when you want to deploy Kafka in your production environment.
- Capacity planning helps you achieve the desired performance from Kafka systems along with the required hardware.
- In this section, we will talk about some of the important aspects to consider while performing capacity planning of Kafka cluster.

Capacity planning goals

- Generally, capacity planning goals are driven by latency and throughput.
- Some of the additional goals could be fault tolerance and high availability.

Replication factor

- Replication factor is one of the main factors in capacity planning.
- As a rule of thumb, one single broker can only host only one partition replica.
- If that had not been the case, one broker failure could have caused Kafka to become unavailable.

Memory

- Kafka is highly dependent on the file system for storing and caching messages.
- All the data is written to the page cache in the form of log files, which are flushed to disk later.
- Generally, most of the modern Linux OS use free memory for disk cache.
- Kafka ends up utilizing 25-30 GB of page cache for 32 GB memory.

Based on that, you can calculate your memory needs using this formula:

Throughput * buffer time

Hard drives

- You should try to estimate the amount of hard drive space required per broker along with the number of disk drives per broker.
- Multiple drives help in achieving good throughput.
- You should also not share Kafka data drives with Kafka logs, Zookeeper data, or other OS file system data, This ensures good latency.
- Based on these estimates, you can calculate the space per broker using the following formula:

(Message Size * Write Throughput * Message Retention Period * Replication Factor) / Number of Brokers

Network

- Kafka is a distributed messaging system, The network plays a very important role in a distributed environment.
- A bad network design may affect the performance of the overall cluster.
- A fast and reliable network ensures that nodes can communicate with each other easily.

CPU

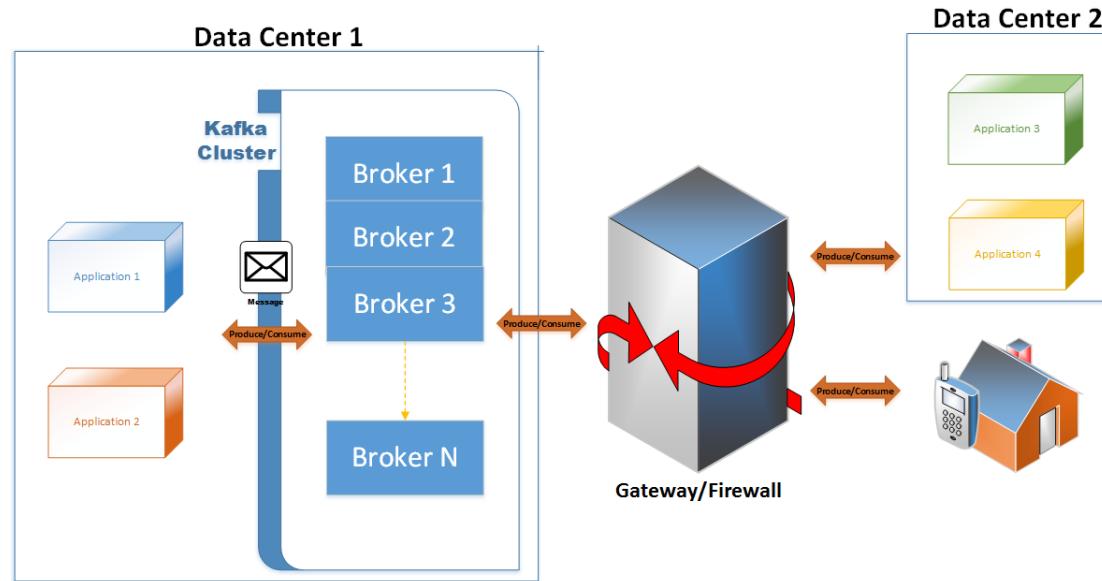
- Kafka does not have very high CPU requirement. Although more CPU cores are recommended.
- Choose a next-generation processor with more cores.
- Common clusters consist of 24 core machines. CPU cores help you add extra concurrency, and more cores will always improve your performance.

Single cluster deployment

- You would have multiple brokers and Zookeeper servers deployed to serve the requests.
- All those brokers and Zookeepers would be in the same data center within the same network subnet.

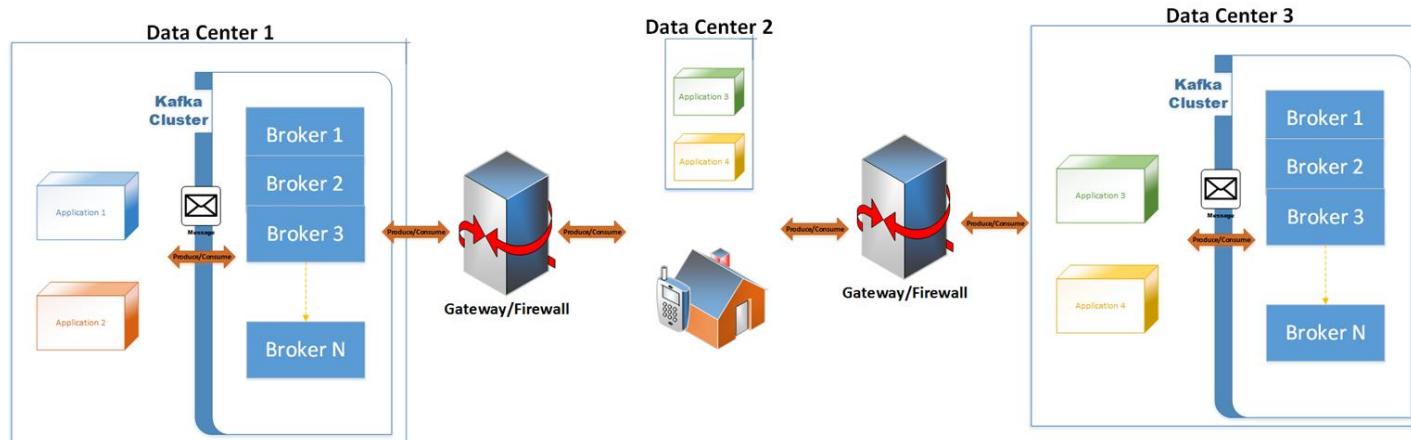
Single cluster deployment

- The following diagram represents what single cluster deployments would look like:



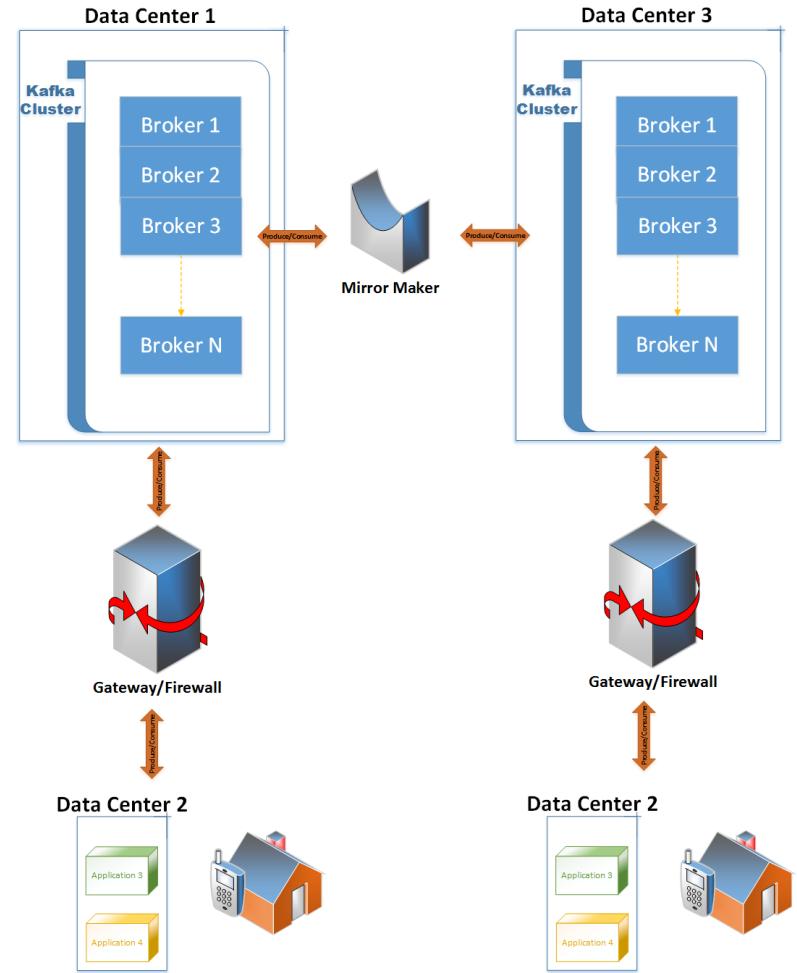
Multicloud deployment

- It's a choice that you must make; whether you want to have routing logic written in your producer or consumer application, or you want to build a separate component to decide on the message routing-based on the Kafka topic.



Multicloud deployment

- Following is a representation of the aggregate model:



Decommissioning brokers

- Kafka is a distributed and replicated messaging system.
- Decommissioning brokers can become a tedious task sometimes.
- In lieu of that, we thought of introducing this section to keep you informed about some of the steps you should perform for decommissioning the broker.

Data migration

- Data migration in Kafka cluster can be viewed in different aspects.
- You may want to migrate data to newly-added disk drives in the same cluster and then decommission old disks.
- You may want to move data to a secure cluster or to newly-added brokers and then decommission old brokers.

Summary

- In this lesson, we dove deep into Kafka cluster. You learned how replication works in Kafka.
- This lesson walked you through how the Zookeeper maintains its znodes and how Kafka uses Zookeeper servers to ensure high availability.
- In this lesson, we wanted you to understand how different processes work in Kafka and how they are coordinated with different Kafka components.

Complete lab 10

11. Using Kafka in Big Data Applications

A blurred background image of a person's hands typing on a laptop keyboard, set against a dark grey diagonal overlay.

Using Kafka in Big Data Applications

The following topics will be covered in the lesson:

- Managing high volumes in Kafka
- Kafka message delivery semantics
- Failure handling and retry-ability
- Big data and Kafka common usage patterns
- Kafka and data governance
- Alerting and monitoring
- Useful Kafka matrices

Managing high volumes in Kafka

In a nutshell, when we talk about high volumes in Kafka, you have to think of following aspects:

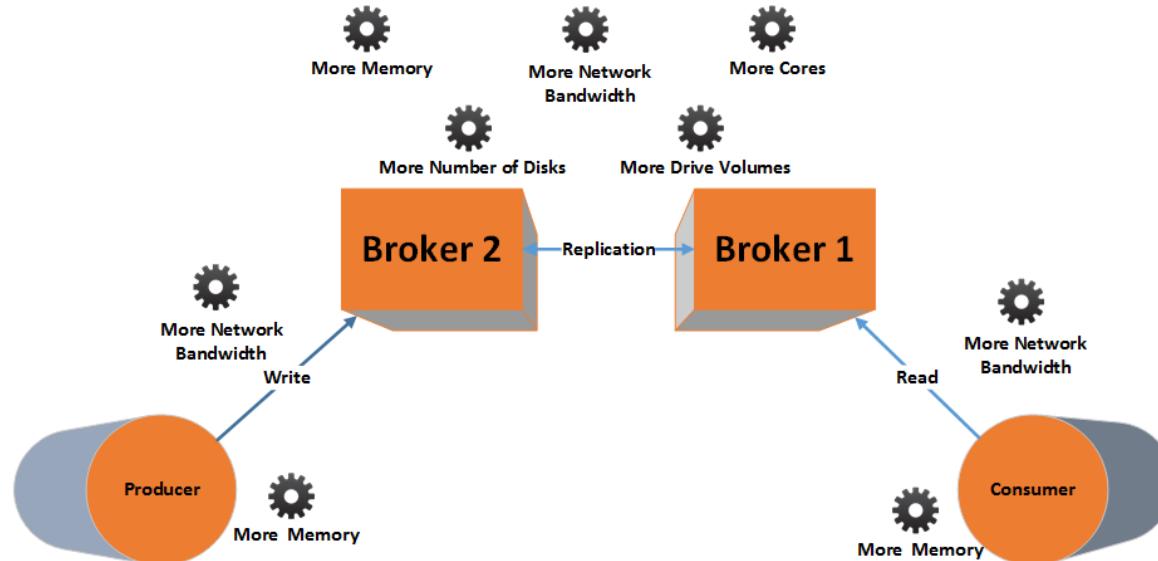
- High volume of writes or high message writing throughput
- High volumes of reads or high message reading throughput
- High volume of replication rate
- High disk flush or I/O

Appropriate hardware choices

- Kafka is a commodity hardware run tool. In cases where volumes are very high, you should first have a clear understanding of which Kafka components are affected and which one of them would need more hardware.

Appropriate hardware choices

- The following diagram will help you understand some of the hardware aspects in case of high volumes:



Producer read and consumer write choices

We are listing some of the techniques that you can use while writing or reading data:

- Message compression
- Message batches
- Asynchronous send
- Linger time
- Fetch size

Kafka message delivery semantics

Semantic guarantees in Kafka need to be understood from the perspective of producers and consumers.

- For example, suppose the producer component does not receive successful acks from brokers because of network connectivity.

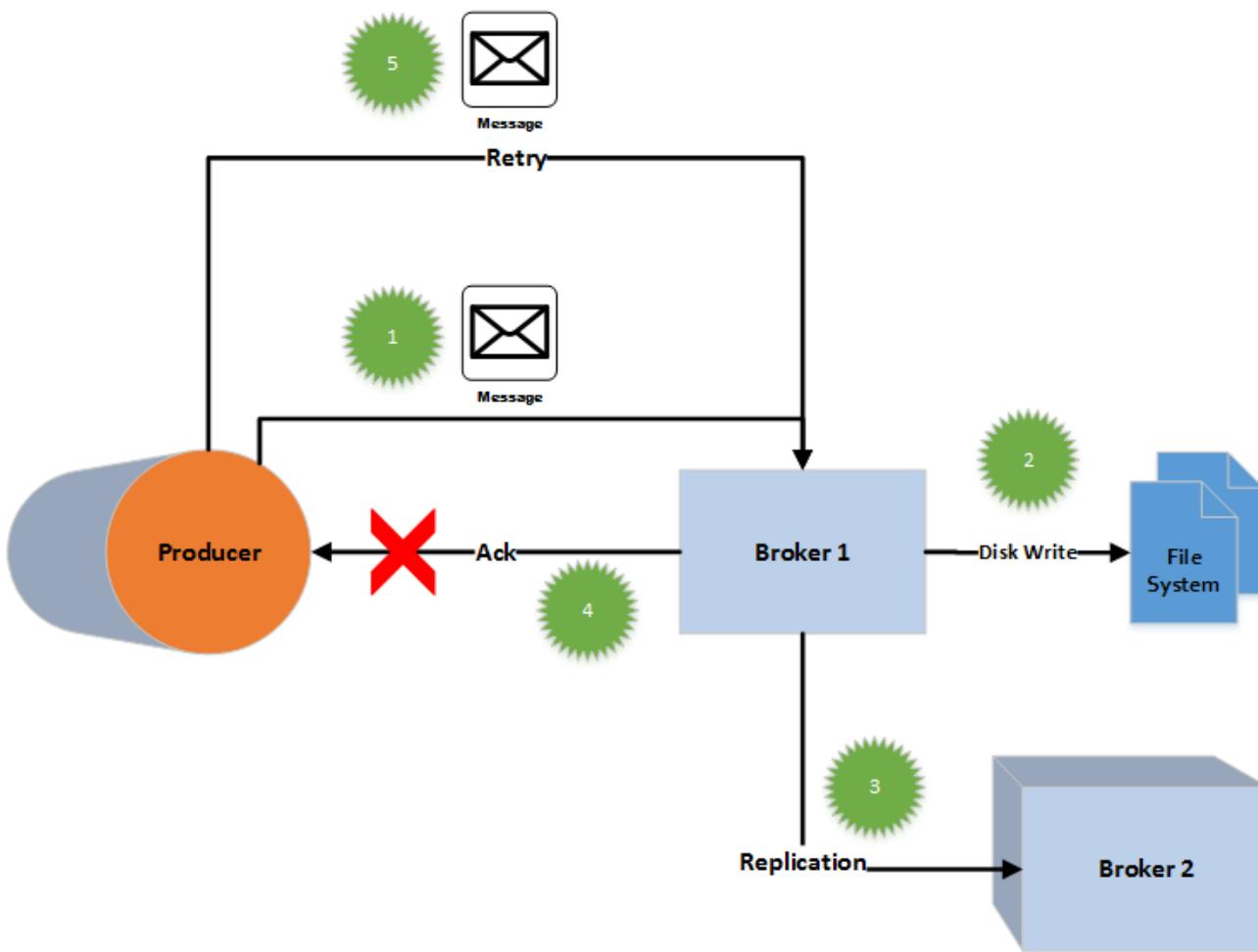
Kafka message delivery semantics

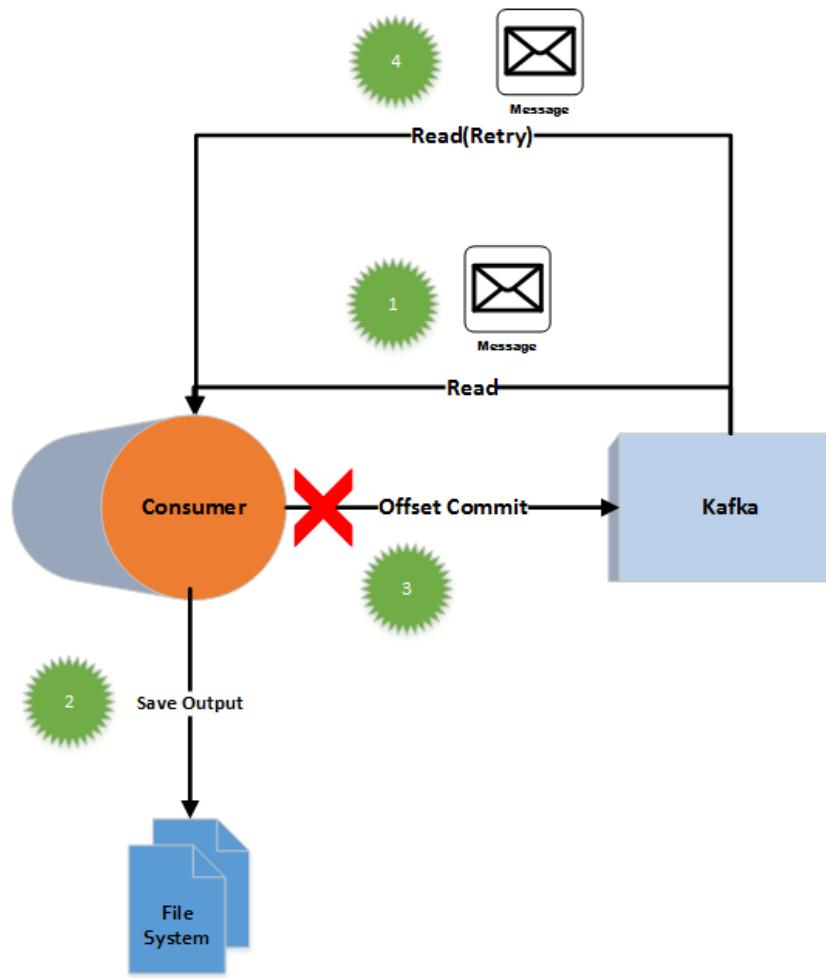
In general, there are three types of message delivery semantics. They are as follows:

- At most once
- At least once
- Exactly Once

At least once delivery

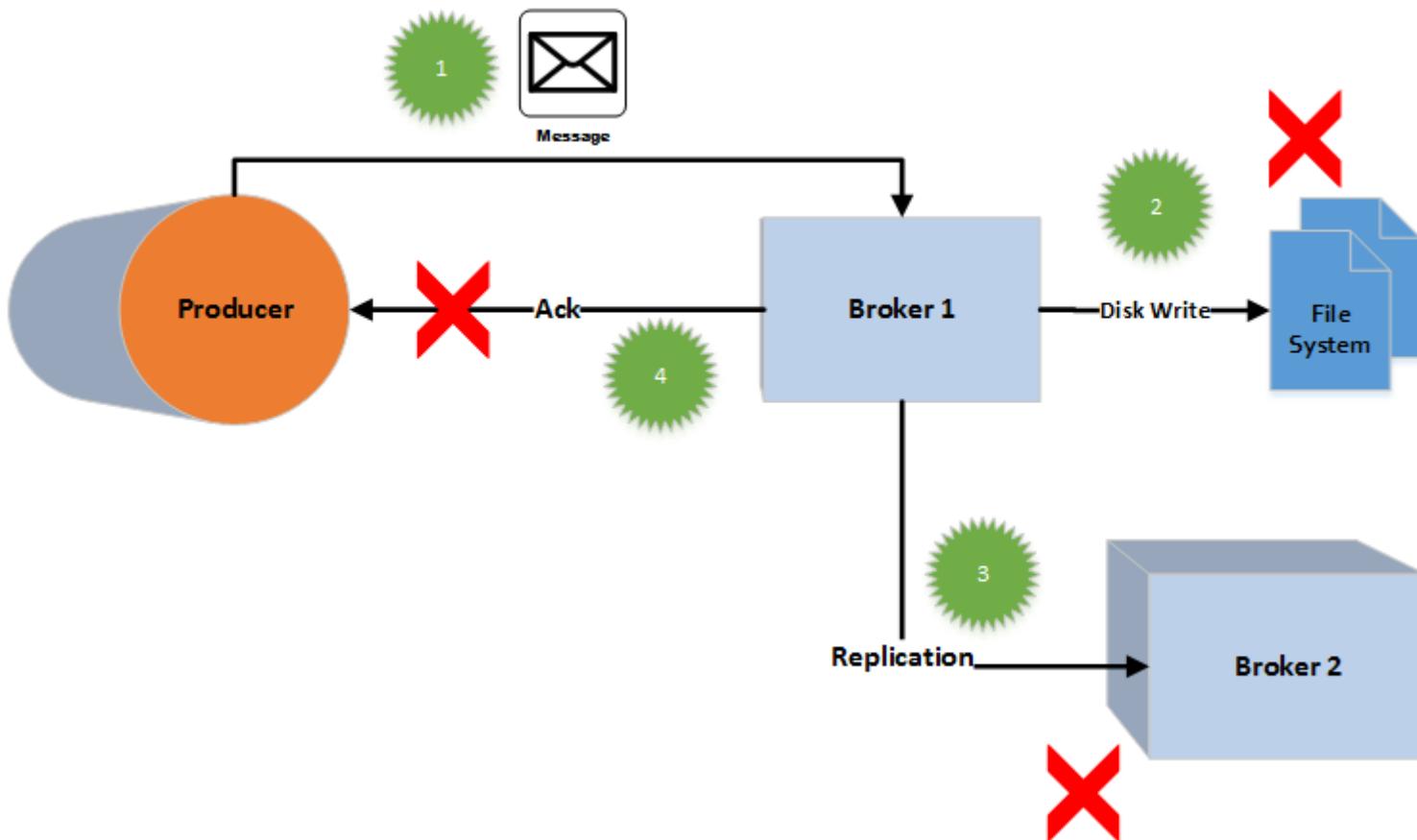
- In the producer context, at least once delivery can happen if acks are lost in network translation.
- Suppose the producer has configuration of acks=all.
- This means the producers will wait for success or failure acknowledgement from the brokers after messages are written and replicated to relevant brokers.
- In case of timeout or some other kind of error, the producer re-sends those messages assuming that they are not written to topic partitions.

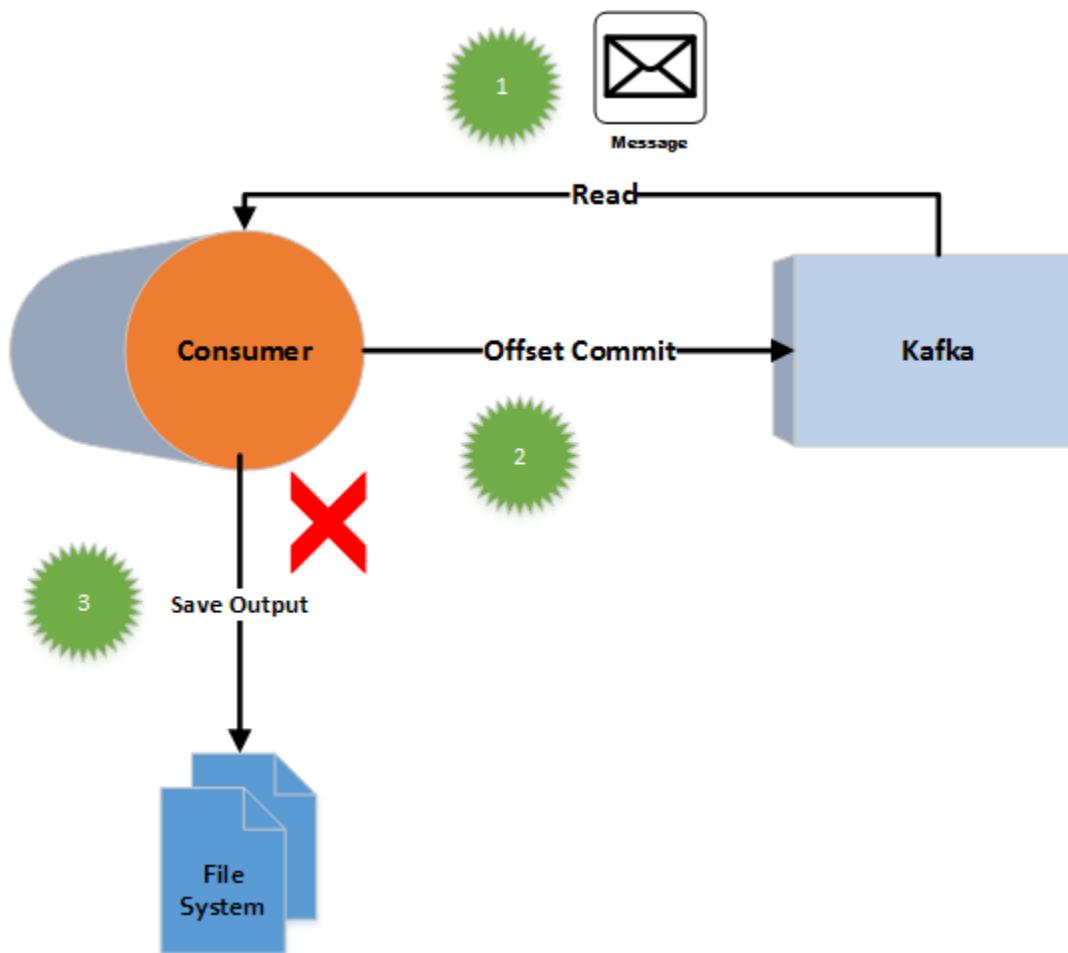




At most once delivery

- In the producer context, at most delivery can happen if the broker has failed before receiving messages or acks are not received and the producer does not try sending the message again.
- In that case, messages are not written to Kafka topic and, hence, are not delivered to the consumer processes, This will result in message loss.





Exactly once delivery

Exactly once delivery needs to be understood in the complete messaging system and not only in the producer or consumer context.

- To ensure exactly once delivery, Kafka has provisions for idempotent producers.
- These kinds of producers ensure that one, and only one, message is written to a Kafka log.

Exactly once delivery

On the consumer side, you have two options for reading transactional messages, expressed through the `isolation.level` consumer config:

- `read_committed`: In addition to reading messages that are not part of a transaction, this allows reading the ones that are, after the transaction is committed.
- `read_uncommitted`: This allows reading all messages in the offset order without waiting for transactions to be committed. This option is similar to the current semantics of a Kafka consumer.

Big data and Kafka common usage patterns

- In the big data world, Kafka can be used in multiple ways.
- One of the common usage patterns of Kafka is to use it as a streaming data platform.
- It supports storing streaming data from varied sources, and that data can later be processed in real time or in batch.

- The following diagram shows a typical pattern for using Kafka as a streaming data platform:

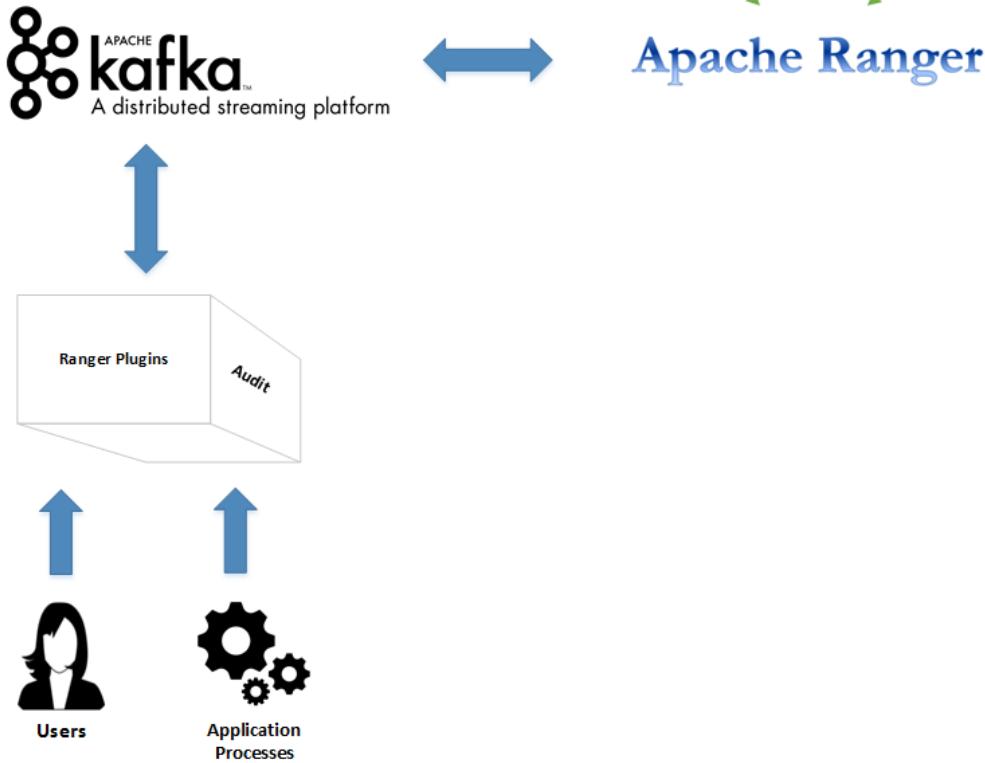


Kafka and data governance

- In any enterprise grade Kafka deployment, you need to build a solid governance framework to ensure security of confidential data along with who is dealing with data and what kind of operations are performed on data.
- Moreover, governance framework ensures who can access what data and who can perform operations on data elements.



- The following diagram represents how data governance can be applied in Kafka using Apache Atlas and Ranger:



Alerting and monitoring

- If you have properly configured the Kafka cluster and it is functioning well, it can handle a significant amount of data.
- If you have Kafka as a centralized messaging system in your data pipeline and many applications are dependent on it, any cluster disaster or bottleneck in the Kafka cluster may affect the performance of all application dependent on Kafka.

Alerting and monitoring

Let's discuss some advantages of monitoring and alerting:

- Avoid data loss
- Producer performance
- Consumer performance
- Data availability

Useful Kafka matrices

We will look into the matrices of Kafka cluster component in detail. The matrices are as follows:

- Kafka producer matrices
- Kafka broker matrices
- Kafka consumer matrices

Producer matrices

Producers are responsible for producing data to Kafka topics. If the producer fails, the consumer will not have any new messages to consume and it will be left idle. The performance of the producer also plays an important role in achieving high throughput and latency. Let's look into a few important matrices of Kafka producer:

- Response rate
- Request rate
- I/O wait time
- Failed send rate
- Buffer total bytes
- Compression rate

Broker matrices

Metrics	Description
kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions	This represents the number of under-replicated partitions. A higher number of under-replication partition may result in losing more data in case the broker fails.
kafka.controller:type=KafkaController, name=OfflinePartitionsCount	This represents the total number of partitions that are not available for read or write because of no active leader for those partitions.
kafka.controller:type=KafkaController, name=ActiveControllerCount	This defines the number of active controllers per cluster. There should not be more than one active controller per cluster.
kafka.server:type=ReplicaManager, name=PartitionCount	This represents the number of partitions on the broker. The value should be even across all brokers.
kafka.server:type=ReplicaManager, name=LeaderCount	This represents the number of leaders on the broker. This should also be even across all brokers; if not, we should enable auto rebalancer for the leader.

Consumer metrics

Consumers are responsible for consuming data from topic and doing some processing on it, if needed. Sometimes, your consumer may be slow, or it may behave unacceptably. The following are some important metrics that will help you identify some parameters that indicate optimization on the consumer side:

- records-lag-max
- bytes-consumed-rate
- records-consumed-rate
- fetch-rate
- fetch-latency-max

Summary

- We walked you through some of the aspects of using Kafka in big data applications.
- By the end of this lesson, you should have clear understanding of how to use Kafka in big data Applications.
- Volume is one of the important aspects of any big data application.

Complete lab 11

12. Securing Kafka

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or professional environment.

Securing Kafka

We will cover the following topics:

- An overview of securing Kafka
- Wire encryption using SSL
- Kerberos SASL for authentication
- Understanding ACL and authorization
- Understanding Zookeeper authentication
- Apache Ranger for authorization
- Best practices for Kafka security

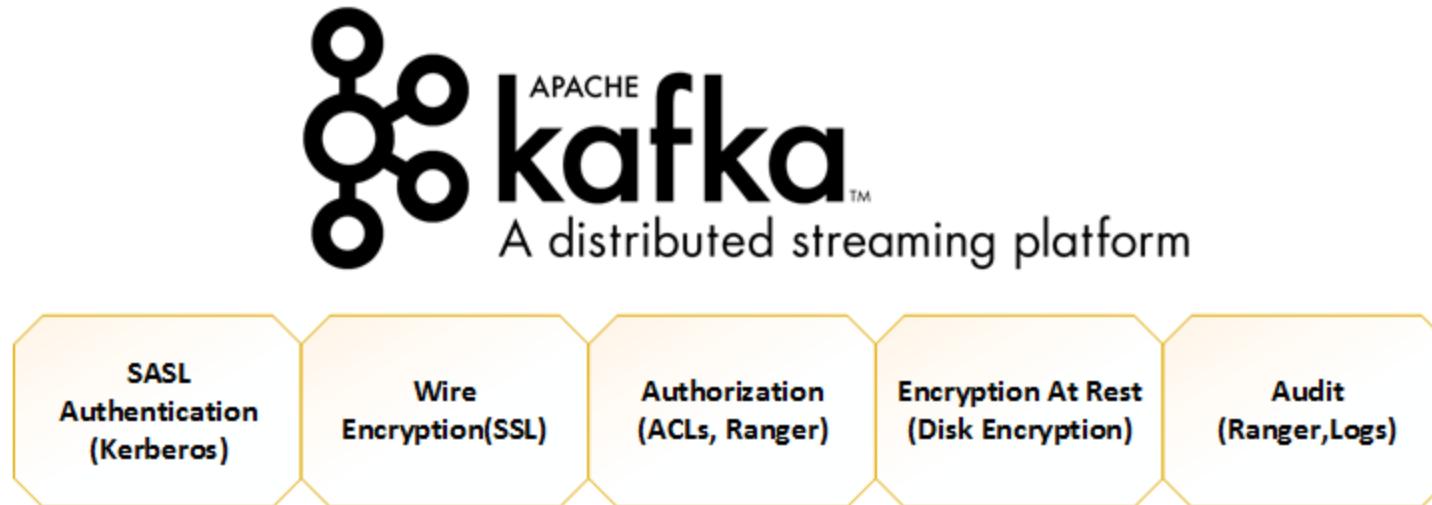
An overview of securing Kafka

In any enterprise deployment of Kafka, security should be looked at from five paradigms. They are as follows:

- Authentication
- Authorization
- Wire encryption
- Encryption at rest
- Auditing

An overview of securing Kafka

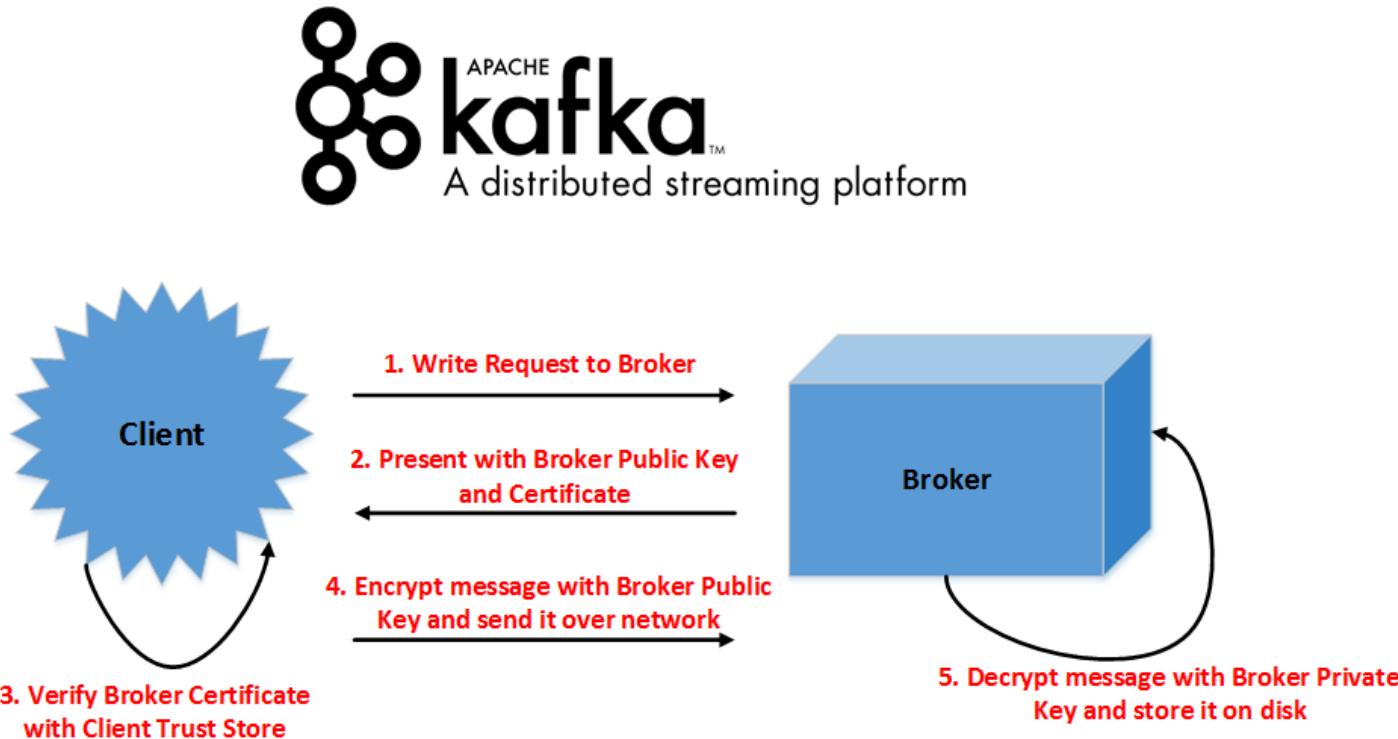
- The following diagram summarizes the different Kafka security paradigms:



Wire encryption using SSL

- In Kafka, you can enable support for Secure Sockets Layer (SSL) wire encryption.
- Any data communication over the network in Kafka can be SSL-wire encrypted.
- Therefore, you can encrypt any communication between Kafka brokers (replication) or between client and broker (read or write).

- The following diagram represents how SSL encryption works in Kafka:



Steps to enable SSL in Kafka

- Let's now look into the steps to enable SSL in Kafka. Before you begin, you should generate the key, SSL certificate, keystore, and truststore that will be used by Kafka clients and brokers.
- You can follow the link
https://kafka.apache.org/documentation/#security_ssl_key to create broker keys and certificate, the link
https://kafka.apache.org/documentation/#security_ssl_ca to create your own certificate authority, and the link
https://kafka.apache.org/documentation/#security_ssl_signing to sign the certificates.

Configuring SSL for Kafka Broker

- To enable SSL for communications between brokers, make the following changes in the broker properties:
security.inter.broker.protocol = SSL
- To configure communication protocols and set SSL ports, make the following changes in server properties:

listeners=SSL://host.name1:port,SSL://host.name2:port

Configuring SSL for Kafka Broker

- To give SSL keystore and truststores path for each broker, you should make the following changes in the server properties of each broker:

ssl.keystore.location =

/path/to/kafka.broker.server.keystore.jks

ssl.keystore.password = keystore_password

ssl.key.password = key_password

ssl.truststore.location =

/path/to/kafka.broker.server.truststore.jks

ssl.truststore.password = truststore_password

Configuring SSL for Kafka clients

- The configuration properties for Kafka producer and consumer are the same.
- The following are the configuration properties you need to set for enabling SSL. If client authentication is not required (ssl.client.auth = none), you need to set the following properties:

`security.protocol = SSL`

`ssl.truststore.location = /path/to/kafka.client.truststore.jks`

`ssl.truststore.password = trustore_password`

Configuring SSL for Kafka clients

- If client authentication is required (ssl.client.auth = required), you need to set the following properties:

security.protocol = SSL

ssl.truststore.location =

/path/to/kafka.client.truststore.jks

ssl.truststore.password = trustore_password

ssl.keystore.location = /path/to/kafka.client.keystore.jks

ssl.keystore.password = keystore_password

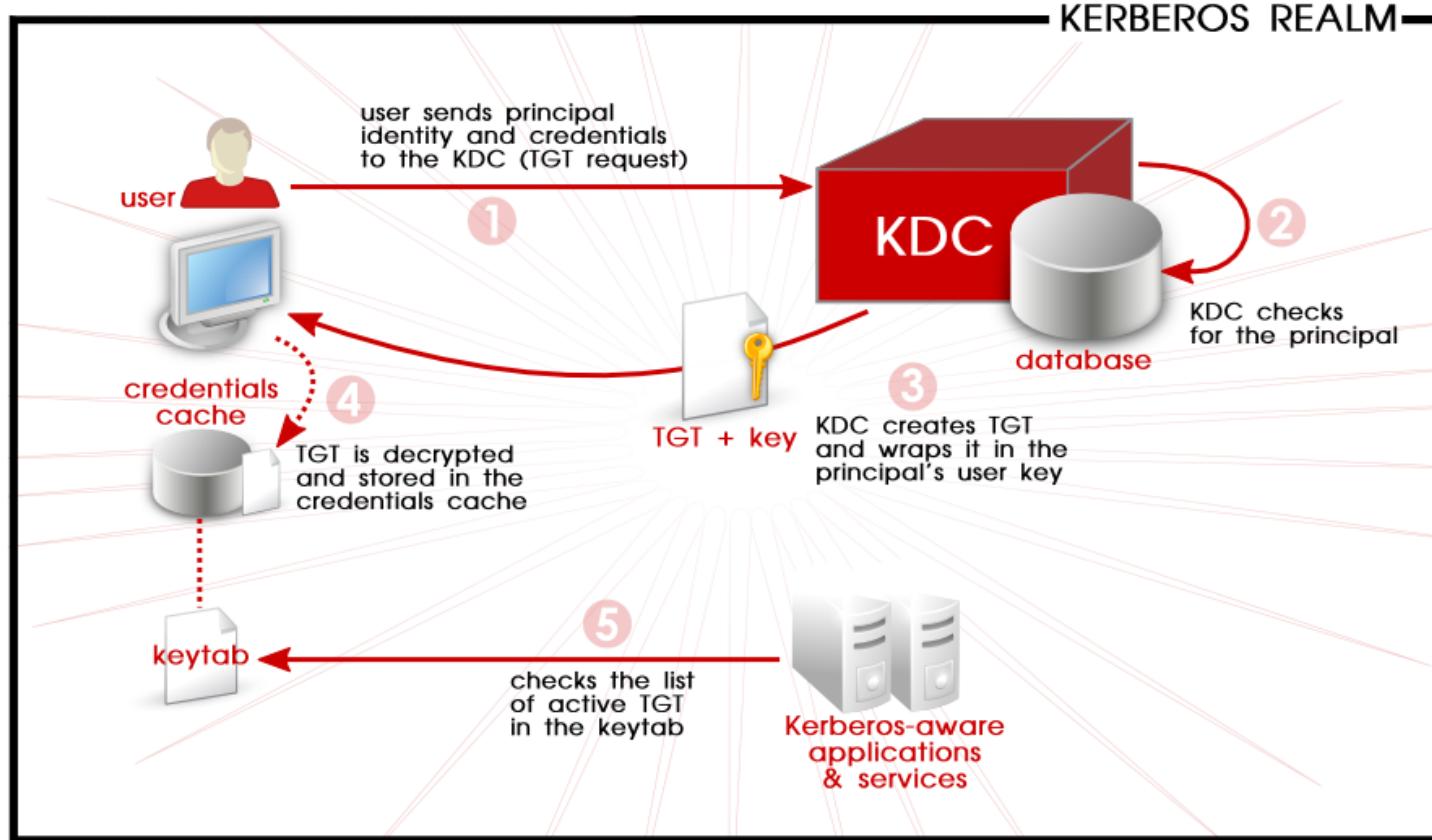
ssl.key.password = key_password

Kerberos SASL for authentication

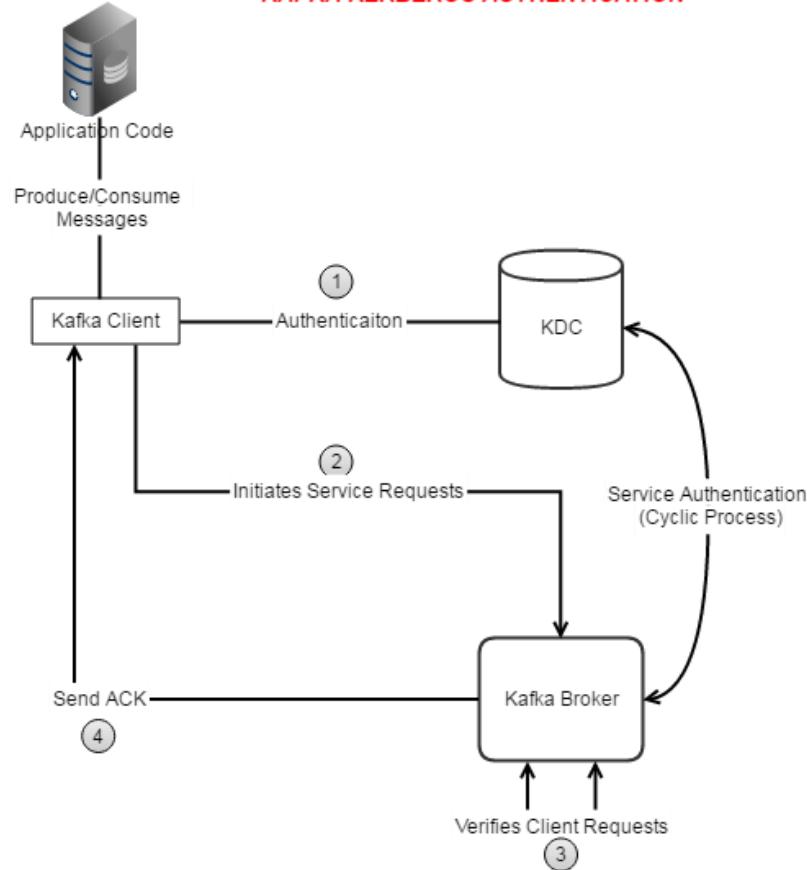
Kerberos is an authentication mechanism of clients or servers over secured network. It provides authentication without transferring the password over the network. It works by using time-sensitive tickets that are generated using symmetric key cryptography. It was chosen over the most-widely-used SSL-based authentication. Kerberos has the following advantages:

- Better performance
- Easy integration with Enterprise Identity Server
- Simpler user management
- No passwords over the network
- Scalable

- The following diagram represents how Kerberos authentication works:



KAFKA KERBEROS AUTHENTICATION



Steps to enable SASL/GSSAPI - in Kafka

Configuring SASL for Kafka broker

- Here is how to configure SASL for Kafka broker:
- Create JAAS configuration files for each broker server, using the following for the content of JAAS files:

Refer to the file 12_1.txt

Steps to enable SASL/GSSAPI - in Kafka

- Once you have saved JAAS configuration to a specific location, you can pass the JAAS file location to each broker's JAVA OPTS as shown in the following code:
 - `Djava.security.auth.login.config=/path/to/kafka_broker_jaas.conf`

Steps to enable SASL/GSSAPI - in Kafka

- Make the following changes into the broker server.properties files.
- If you have SSL enabled in Kafka, make the following property file changes:

```
listeners=SASL_SSL://broker.host.name:port  
advertised.listeners=SASL_SSL://broker.host.name:port  
security.inter.broker.protocol=SASL_SSL  
sasl.mechanism.inter.broker.protocol=GSSAPI  
sasl.enabled.mechanisms=GSSAPI  
sasl.kerberos.service.name=kafka
```

Steps to enable SASL/GSSAPI - in Kafka

- If you do not have SSL enabled in Kafka, make following property file changes:

```
listeners=SASL_PLAINTEXT://broker.host.name:port  
advertised.listeners=SASL_PLAINTEXT://broker.host.n  
ame:port  
security.inter.broker.protocol=SASL_PLAINTEXT  
sasl.mechanism.inter.broker.protocol=GSSAPI  
sasl.enabled.mechanisms=GSSAPI  
sasl.kerberos.service.name=kafka
```

Configuring SASL for Kafka client - producer and consumer

- The first step you should perform is to create JAAS configuration files for each producer and consumer application.
- Use the following for the content of the JAAS files:
`sasl.jaas.config=com.sun.security.auth.module.Krb5Log
inModule required
useKeyTab=true
storeKey=true
keyTab="/path/to/kafka_client.keytab"
principal="kafka-client@REALM";`

Configuring SASL for Kafka client - producer and consumer

- The mentioned JAAS configuration is for Java processes or for applications acting as producer or consumer.
- If you want to use SASL authentication for command line tools, use the following configurations:

```
KafkaClient {  
    com.sun.security.auth.module.Krb5LoginModule  
    required  
    useTicketCache=true;};
```

Configuring SASL for Kafka client - producer and consumer

- Once you have saved JAAS configuration to specific location, you can pass the JAAS file location to each client's JAVA OPTS as shown here:
 - `Djava.security.auth.login.config=/path/to/kafka_client_ja
as.conf`

Configuring SASL for Kafka client - producer and consumer

- Make the following changes to the producer.properties or consumer.properties files.
- If you have SSL enabled in Kafka, make the following property file changes:

security.protocol=SASL_SSL
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka

Configuring SASL for Kafka client - producer and consumer

- If you do not have SSL enabled in Kafka, make the following property file changes:

```
security.protocol=SASL_PLAINTEXT  
sasl.mechanism=GSSAPI  
sasl.kerberos.service.name=kafka
```

Understanding ACL and authorization

Principal P is Allowed OR Denied Operation O From Host H On Resource R.

The terms used in this definition are as follows:

- Principal is the user who can access Kafka
- Operation is read, write, describe, delete, and so on
- Host is an IP of the Kafka client that is trying to connect to the broker
- Resource refers to Kafka resources such as topic, group, cluster

Understanding ACL and authorization

Let's discuss a few common ACL types:

- Broker or server ACL
- Topic
- Producer
- Consumer

Common ACL operations

- Kafka provides a simple authorizer; to enable this authorizer, add the following line to server properties of Kafka:

authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer

Common ACL operations

- As discussed in previous paragraphs, by default, only a superuser will have access to resources if no ACL is found.
- However, this behavior can be changed if we want to allow everyone to access resources if no ACL is set.
- Add the following line to server properties:

`allow.everyone.if.no.acl.found=true`

Common ACL operations

- You can also add more superusers to your Kafka cluster by adding users to the following property in the server property file:

super.users=User:Bob;User:Alice

Common ACL operations

```
kafka-acls.sh --authorizer
kafka.security.auth.SimpleAclAuthorizer --authorizer-
properties zookeeper.connect=localhost:2181 --add --
allow-principal User:Chanchal --allow-principal
User:Manish --allow-hosts
10.200.99.104,10.200.99.105 --operations Read,Write --
topic fenago
```

Common ACL operations

- Removing ACL: The ACL added in the preceding part can be removed using the following command:

```
kafka-acls.sh --authorizer
```

```
kafka.security.auth.SimpleAclAuthorizer --authorizer-  
properties zookeeper.connect=localhost:2181 --remove  
--allow-principal User:Chanchal --allow-principal  
User:Manish --allow-hosts  
10.200.99.104,10.200.99.105--operations Read,Write --  
topic fenago
```

List ACLs

- For example, if you want to see all ACLs applied in the topic fenago, you can do it using the following command:

```
kafka-acls.sh --authorizer  
kafka.security.auth.SimpleAclAuthorizer --authorizer-  
properties zookeeper.connect=localhost:2181 --list --  
topic fenago
```

List ACLs

- Producer and consumer ACL: Adding a user as the producer or consumer is a very common ACL used in Kafka.
- If you want to add user Chanchal as a producer for topic fenago, it can be done using the following simple command:

```
kafka-acls --authorizer-properties  
zookeeper.connect=localhost:2181\  
--add --allow-principal User:Chanchal \  
--producer --topic fenago
```

List ACLs

- To add a consumer ACL where Manish will act as the consumer for topic fenago with consumer group G1, the following command will be used:

```
kafka-acls --authorizer-properties  
zookeeper.connect=localhost:2181\  
--add --allow-principal User:Manish \  
--consumer --topic fenago --group G1
```

Understanding Zookeeper authentication

- Zookeeper is the metadata service for Kafka. SASL-enabled Zookeeper services first authenticate access to metadata stored in Zookeeper.
- Kafka brokers need to authenticate themselves using Kerberos to use Zookeeper services.
- If valid, the Kerberos ticket is presented to Zookeeper, it then provides access to the metadata stored in it.

Understanding Zookeeper authentication

- One is using Kerberos key tabs. An example of such login context can be seen as follows:

```
Client {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    keyTab="/path/to/client/keytab(Kafka keytab)"  
    storeKey=true  
    useTicketCache=false  
    principal="yourzookeeperclient(Kafka)";  
};
```

Understanding Zookeeper authentication

- The second one is by user login credential cache. An example of such login context can be seen as follows:

```
Client {  
    com.sun.security.auth.module.Krb5LoginModule  
    required  
        useKeyTab=false  
        useTicketCache=true  
        principal="client@REALM.COM";  
        doNotPrompt=true  
};
```

Apache Ranger for authorization

- Ranger is used to monitor and manage security across the Hadoop ecosystem.
- It provides a centralized platform from which to create and manage security policies across the cluster.
- We will look at how we can use Ranger to create policies for the Kafka cluster.

Adding Kafka Service to Ranger

Ranger Access Manager Audit Settings

Create Service

Service Details :

Service Name *

Description

Active Status Enabled Disabled

Config Properties :

Username *

Password *

Zookeeper Connect String * localhost:2181

Ranger Plugin SSL CName

Add New Configurations

Name	Value
<input type="text"/>	<input type="text"/> *

+

Test Connection

Add **Cancel**

Adding policies

Ranger Access Manager Audit Settings admin

Service Manager > kafkadev Policies

List of Policies : kafkadev

CLICK ON ADD NEW POLICY BUTTON

SEARCH FILTER FOR POLICIES

Policy ID	Policy Name	Status	Audit Logging	Groups	Users	Action
18	kafkadev-1-20150731100352	Enabled	Enabled		mark	<input type="checkbox"/> <input type="checkbox"/>
19	kafka policy	Enabled	Enabled			<input type="checkbox"/> <input type="checkbox"/>

Adding policies

Ranger Access Manager Audit Settings admin

Service Manager > kafka-dev Policies > Create Policy

Create Policy

Policy Details :

Policy Name *	<input type="text"/>	<input checked="" type="radio"/> enabled
Topic *	<input type="text"/>	<input checked="" type="radio"/> include
Description	<input type="text"/>	

Audit Logging YES

User and Group Permissions :

Permissions	Select Group	Select User	Policy Conditions	Permissions	Delegate Admin	X
	<input type="text"/> Select Group +	<input type="text"/> Select User	Add Conditions +	Add Permissions +	<input type="checkbox"/>	X

Add Cancel

Best practices

- In Kafka or, as a matter of fact, in any JAVA Kerberos-enabled application, you can set the Kerberos debug level using the following property:

`sun.security.krb5.debug=true`

Summary

- In this lesson, we covered different Kafka security paradigms.
- Our goal with this lesson is to ensure that you understand different paradigms of securing Kafka. We wanted you to first understand what are different areas you should evaluate while securing Kafka.
- After that, we wanted to address how parts of securing Kafka.

Complete lab 12

13. Streaming Application Design Considerations

Streaming Application Design Considerations

The following topics will be covered in this lesson:

- Latency and throughput
- Data persistence
- Data sources
- Data lookups
- Data formats
- Data serialization
- Level of parallelism
- Data skews
- Out-of-order events
- Memory tuning

Latency and throughput

- One of the fundamental features of any streaming application is to process inbound data from different sources and produce an outcome instantaneously.
- Latency and throughput are the important initial considerations for that desired feature.
- In other words, performance of any streaming application is measured in terms of latency and throughput.

Data and state persistence

- Data integrity, safety, and availability are some of the key requirements of any successful streaming application solution.
- If you give these factors a thought, you will understand that to ensure integrity, safety, and availability, persistence plays an important role.
- For example, it is absolutely essential for any streaming solution to persists its state.

Data sources

- One of the fundamental requirements for any streaming application is that the sources of data should have the ability to produce unbound data in terms of streams.
- Streaming systems are built for unbound data streams.

External data lookups

- The first question that must be in your mind is why we need external data lookups in the stream processing pipeline.
- The answer is that sometimes you need to perform operations such as enrichment, data validation, or data filtering on incoming events based on some frequently changing external system data.
- However, in the streaming design context, these data lookups pose certain challenges.

Data formats

- One of the important characteristics of any streaming solution is that it serves as an integration platform as well.
- It collects events from varied sources and performs processing on these different events to produce the desired outcomes.
- One of the pertinent problems with such integration platforms is different data formats.

Each type of source has its own format.

Data serialization

- Almost all the streaming technology of your choice supports serialization.
- However, key for any streaming application performance is the serialization technique used.
- If the serialization is slow, then it will affect your streaming application latency.

Level of parallelism

- Any stream processing engine of your choice has ways to tune stream processing parallelism.
- You should always give a thought to the level of parallelism required for your application.
- A key point here is that you should utilize your existing cluster to its maximum potential to achieve low latency and high throughput.

Out-of-order events

- This is one of the key problems with any unbound data stream.
- Sometimes an event arrives so late that events that should have been processed after that out of order event are processed first.
- Events from varied remote discrete sources may be produced at the same time and, due to network latency or some other problem, some of them are delayed.

Message processing semantics

- Exactly-once delivery is the holy grail of streaming analytics.
- Having duplicates of events processed in a streaming job is inconvenient and often undesirable, depending on the nature of the application.
- For example, if billing applications miss an event or process an event twice, they could lose revenue or overcharge customers.

Summary

- At the end of this lesson, you should have a clear understanding of various design considerations for streaming applications.
- Our goal with this lesson was to ensure that you have understood various complex aspects of a streaming application design.
- Although the aspects may vary from project to project, based on our industry experience.

Complete lab 13